Technical Report
CMU/SEI-93-TR-27
ESC-TR-93-201

Carnegie-Mellon University
Software Engineering Institute

A Survey of Commonly Applied Methods for Software Process Improvement

Robert D. Austin
Daniel J. Paulish

February 1993

Technical Report
CMU/SEI-93-TR-27
ESC-TR-93-201
February 1994

A Survey of Commonly Applied Methods for Software Process Improvement

Robert D. Austin
Doctoral Candidate
Carnegie Mellon University

Daniel J. Paulish
Resident Affiliate
Siemens Corporate Research, Inc.

Software Process Measurement Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESC/ENS
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.
This report was funded by the U.S. Department of Defense.

Table of Contents

List of Figures

List of Tables

Preface

This report is an output of a joint Software Engineering Institute (SEI)/Siemens project in which Siemens software development organizations are being used as case study sites to measure and observe the impact of methods used to improve the software development process. The project will result in the identification of specific interventions that can be tailored to the current maturity level of an organization that wishes to improve. The project is intended to assist software development managers in selecting from the commonly applied software process improvement methods.

This report describes methods that are commonly used within industrial organizations for improving the software development process. The report can be used by software engineering managers who are considering alternative actions for improving their development process. It could also prove useful to change agents of software process improvement, such as found within a software engineering process group (SEPG), and practitioners such as software engineers and quality specialists. The reader of this report should learn about a number of specific process improvement methods including key references for obtaining further information about the application and experiences of other organizations with the described method.

The report describes the methods in the context of the Capability Maturity Model (CMM). The methods are correlated to key process areas (KPAs) of the CMM so that organizations at different levels of maturity can better identify the best methods to select and implement.

Each process improvement method is concisely described from surveying existing technical literature citations. Each method description contains background material concerning its origin, how it works, documented experience with its application, suggestions for introduction and use, how it relates to the CMM, and a list of key references for further information.

A Survey of Commonly Applied Methods for Software Process Improvement

Abstract: This report describes a number of commonly applied methods for improving the software development process. Each software process improvement method is described by surveying existing technical literature citations. Each method description contains background information concerning how the method works. Documented experience with the method is described. Suggestions are given for implementing the method, and a list of key references is given for further information. The methods are described in the context of the SEI Capability Maturity Model, and suggestions are given to assist organizations in selecting potential improvement methods based upon their current process maturity.

1 Introduction

1.1 Motivation for Software Process Improvement

Many software engineering organizations today have the desire to improve their software development process as a way of improving product quality and development team productivity and reducing product development cycle time, thereby increasing business competitiveness and profitability. Although many organizations are motivated to improve, very few organizations know how best to improve their development process. There is a wide assortment of available methods such as total quality management (TQM), quality function deployment (QFD), function point analysis (FPA), defect prevention process (DPP), software quality assurance (SWQA), configuration management (CM), software reliability engineering (SRE), etc. This often creates confusion for software engineering managers with respect to which methods should be introduced at which point within their process evolution (Figure 1-1).

The motivation to improve a software process usually results from a business need such as strong competition, increased profitability, or external regulation. Approaches to improve a software development process, such as shown in Figure 1-2, are often initiated by an assessment of the current practices and maturity. A number of improvement methods are then recommended and implemented. The selection and successful implementation of the best improvement methods are dependent on many variables such as the current process maturity, skills base, organization, and business issues such as cost, risk, implementation speed, etc. Measuring the impact and predicting the success of a specific improvement method are difficult. This is often due to environmental variables external to the method such as staff skills, acceptance, training effectiveness, and implementation efficiency. Once the improvement method is in place, there is also the question of what to do next. It is necessary to determine whether the method was implemented successfully, whether the process is mature enough to consider implementing additional methods, or whether the selected method is appropriate for use within the current process maturity level and environment.

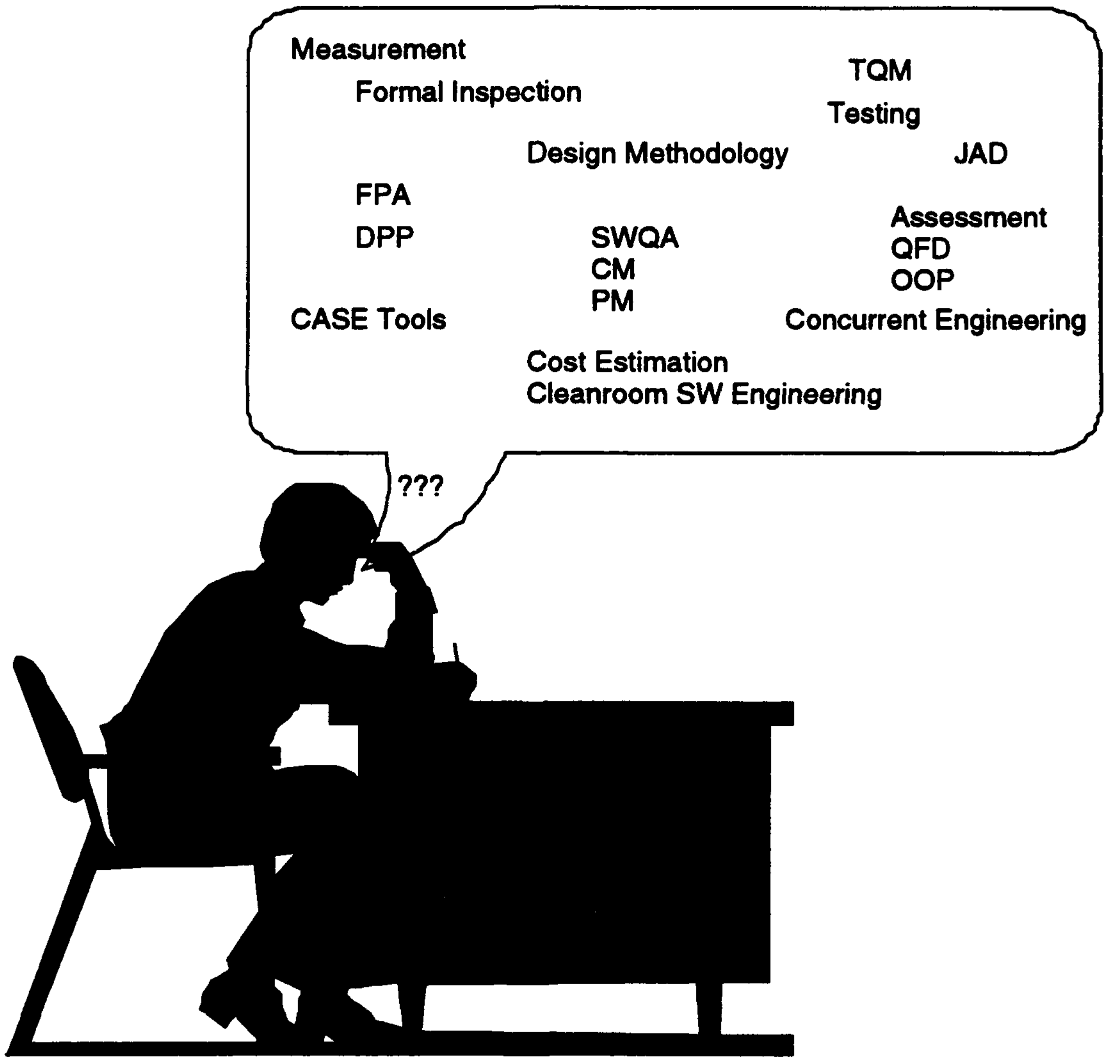

Figure 1-1. Which Software Process Improvement Methods?

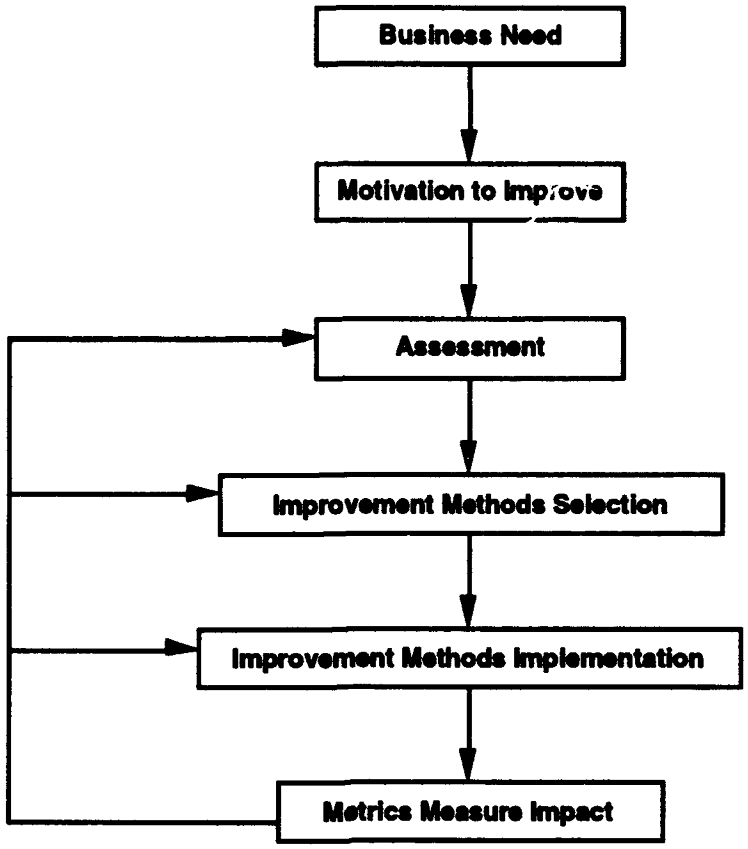

Figure 1-2. Process Improvement Approach

1.2 Purpose and Scope

This report describes methods for improving the software process related to key practices of the SEI Capability Maturity Model (CMM). Practical suggestions concerning the implementation and impact of process improvement methods are given to software development organizations in order to provide the foundation for continuous process improvement.

This report describes some process improvement methods that have been commonly applied within industrial software development organizations. All possible improvement methods cannot be adequately described herein; however a subset of commonly applied methods has been selected to give software engineering managers some background and guidance on methods that have been successfully applied in other organizations. For this report, we have selected methods that adhere to the definition given above and that often require significant training and effort to introduce to a software development organization. Thus, the methods selected usually require an investment by the organization. There may often be barriers which must be overcome for adoption of the method in the organization before one can observe a measurable impact resulting from the improved software process. Particular attention has been paid in the literature survey to experience reports that claim a measurable performance improvement as a result of applying the method. The experience ranges from anecdotal evidence to controlled experiments.

This report contains descriptions of the following software process improvement methods: estimation, function point analysis (FPA), ISO 9000 certification, software process assessment (SPA), software engineering process group (SEPG), process definition, formal inspection, software measurement and metrics, computer aided software engineering (CASE), interdisciplinary group methods (IGMs), nominal group technique (NGT), joint application design (JAD), groupware, group decision support systems (GDSSs), quality circles (QCs), concurrent engineering, software reliability engineering (SRE), quality function deployment (QFD), total quality management (TQM), defect prevention process (DPP), and cleanroom software development.

It is planned that future versions of this report will contain descriptions of additional widely practiced software process improvement methods. The literature survey will be supported by case studies within industrial organizations in which the application of specific software process improvement methods will be observed, and their impact on organization performance measured over time.

1.3 Related Efforts

This literature survey report is an output of a joint SEI/Siemens project. Within this project a number of Siemens software development organizations have been selected as case study sites to investigate the impact of selected process improvement methods. A limited number of basic measurements have been made to capture the current performance of the development organization with respect to development team productivity, schedule, process maturity, and product quality.

Within the case study sites, a number of process improvement methods have been selected for implementation dependent on the current maturity level, skills base, organization structure, and business issues. The development organizations will be revisited periodically at which time the basic measurements will be recalculated. This will provide some quantitative data concerning the impact of the selected process improvement methods. In addition, lessons learned from the implementation and impact of the process improvement methods will be captured and documented. In particular, we will capture observations concerning the use of the methods including soft factors such as the impact on staff morale, quality culture, motivation, etc.

The case study approach and results will be documented in additional technical reports. For each case study organization we will describe:

- The practices, process, and environment.
- The improvement methods approach.
- The performance measurements.
- The soft factors observed, lessons learned, and improvement results.

Each case study will be concisely described within the technical reports. A small subset of the case studies will be selected for more detailed observation and description as appropriate.

2 What Is a Process Improvement Method?

2.1 Definition

A software process improvement method is defined as an integrated collection of procedures, tools, and training for the purpose of increasing product quality or development team productivity, or reducing development time. Typical results of an improved software development process could include:

- Fewer product defects found by customers.
- Earlier identification and correction of defects.
- Fewer defects introduced during the development process.
- Faster time to market.
- Better predictability of project schedules and resources.

A software process improvement method can be used to support the implementation of a key process area (KPA) of the Capability Maturity Model (CMM) or to improve the effectiveness of key practices within a KPA. The CMM identifies five levels of maturity of a development organization [Paulk 93].

1. Initial. The development environment is unstable. The organization does not consistently apply software engineering management to the process, nor does it use modern tools and technology. Performance can only be predicted by individual, rather than organizational, capability. Level 1 organizations may have serious cost and schedule problems.

2. Repeatable. At level 2, the organization has installed basic software management controls. Stable processes are in place for planning and tracking software projects. Project standards exist and are used.

3. Defined. At level 3, the organization has a standard process for developing and maintaining software across the organization. The software engineering and software management processes are integrated into a coherent whole. A software engineering process group (SEPG) facilitates software process definition and improvement efforts. Organization wide training is in place, to assure that all employees have the skill necessary to perform their duties. Peer reviews are used to enhance product quality.

4. Managed. At level 4, the organization sets quantitative quality goals for software products. Productivity and quality are measured for important software process activities across all projects in the organization. A process database is used to collect and analyze the data from a carefully designed process. There are well-defined and consistent measures for evaluating processes and products.

5. Optimizing. At level 5, the organization is focused on continuous improvement. There are means of identifying weak processes and strengthening them. Statistical evidence is available on process effectiveness and is used in performing cost-benefit analyses on new technologies. Innovations that exploit the best software engineering practices are identified.

The software process improvement methods described in this report and their corresponding primary key process areas of the CMM are given in Table 1.

Table 1: Software Process Improvement Methods

Method	Key Process Area	CMM Level
Estimation	Software project planning	2
ISO 9000 certification	Software quality assurance Organization process def.	2 3
Software process assessment (SPA)	Organization process focus	3
Process definition	Organization process def.	3
Formal inspection	Peer reviews	3
Software measurement & metrics	Software project planning Software project tracking & oversight Integrated software mgt. Quantitative process mgt. Software quality mgt. Process change mgt.	2 2 3 4 4 5

Table 1: Software Process Improvement Methods (Continued)

Method	Key Process Area	CMM Level
Computer aided software engineering (CASE)	Software configuration mgt. Software quality assurance Software project tracking & oversight Organization process def. Software product engineering	2 2 2 3 3
Interdisciplinary group methods (IGMs)	Intergroup coordination	3
Software reliability engineering (SRE)	Quantitative process mgt.	4
Quality function deployment (QFD)	Software quality mgt.	4
Total quality management (TQM)	Organization process focus Quantitative process mgt. Software quality mgt Process change management	3 4 4 5
Defect prevention process (DPP)	Defect prevention	5
Cleanroom software development	Quantitative process mgt. Software quality mgt. Defect prevention	4 4 5

The Capability Maturity Model and some examples of software process improvement methods correlated to the key process areas are given in Figure 2-1.

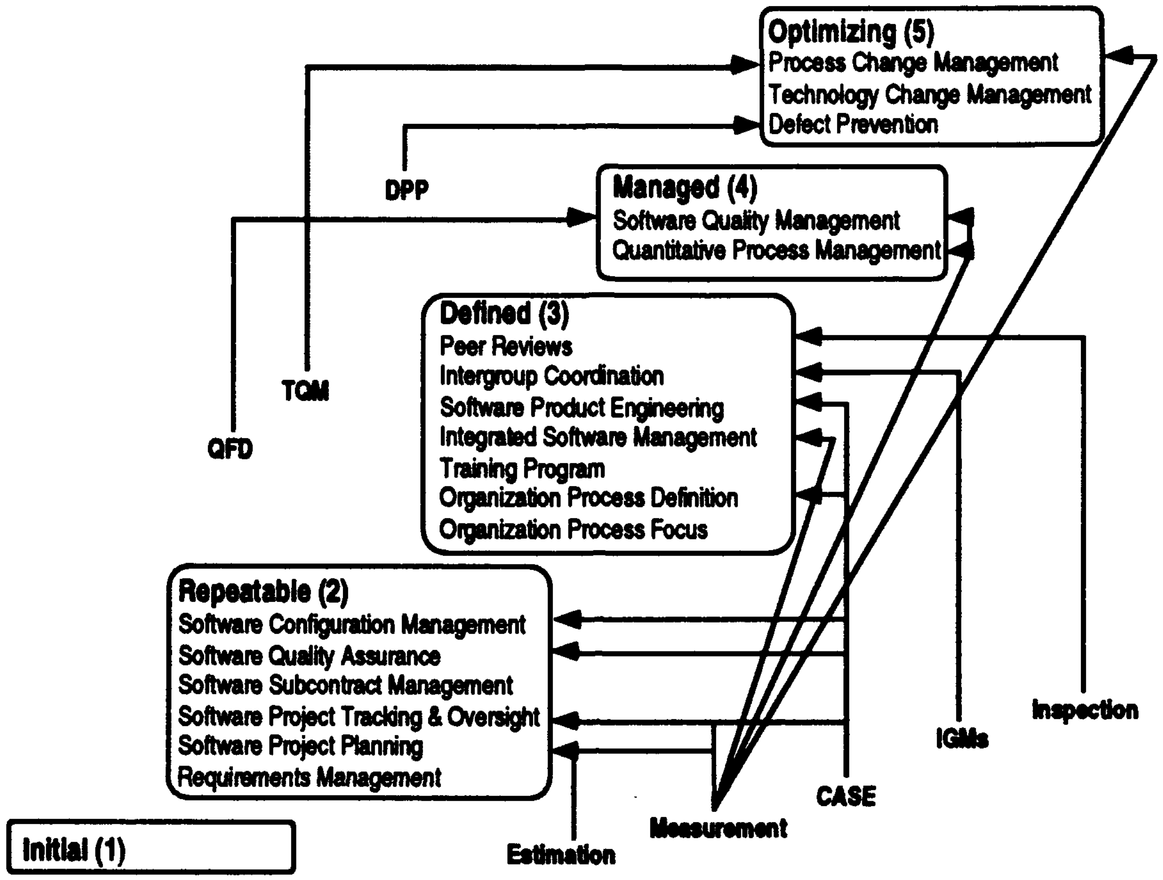
Optimizing (5)
Process Change Management
Technology Change Management
Defect Prevention
Managed (4)
Software Quality Management
Quantitative Process Management
Defined (3)
Peer Reviews
Intergroup Coordination
Software Product Engineering
Integrated Software Management
Training Program
Organization Process Definition
Organization Process Focus
Repeatable (2)
Software Configuration Management
Software Quality Assurance
Software Subcontract Management
Software Project Tracking & Oversight
Software Project Planning
Requirements Management
Initial (1)
DPP
TQM
QFD
Inspection
IGMs
CASE
Measurement
Estimation

Figure 2-1. Example Software Process Improvement Methods

2.2 Report Content

The specific process improvement methods described in this report are summarized below.

- *Estimation:* This collection of methods uses models and tools to predict characteristics of a software project such as schedule and staff resources before the project begins.
- *ISO 9000 certification:* ISO 9000 is a series of quality standards established by the International Standards Organization (ISO) for certifying that an organization's practices meet an acceptable level of quality control.

- *Software process assessment (SPA):* Assessment methods are a means of determining the strengths and weaknesses of an organization's software development process. Results include a "maturity rating," and findings of potential areas for improvement which are often implemented by a software engineering process group (SEPG).
- *Process definition:* These methods refer to the practice of formally specifying or modeling the software development process in a way that allows communication and analysis through its representation.
- *Formal inspection:* This method, pioneered by Michael Fagan at IBM in the 1970s, provides a technique to conduct review meetings to identify defects for subsequent correction within code or other documents.
- *Software measurement and metrics:* This collection of methods provides mechanisms for defining, tracking, and analyzing measures that can be used for controlling and improving the software development process.
- *Computer aided software engineering (CASE):* This collection of methods uses software tools for automating the software development process, particularly in the areas of design and analysis.
- *Interdisciplinary group methods (IGMs):* This collection of methods refers to various forms of planned interaction engaged in by people of diverse expertise and functional responsibilities working together as a team toward the completion of a software system. Example methods include nominal group technique (NGT), joint application design (JAD), groupware, group decision support systems (GDSSs), quality circles (QCs), and concurrent or simultaneous engineering.
- *Software reliability engineering (SRE):* SRE is a collection of methods using models for statistically predicting failure rates of a software system before it is released.
- *Quality function deployment (QFD):* This method is used to assist in defining software functional requirements that can best meet customer needs, distinguishing resulting products from those of competitors, and considering implementation difficulty.

- *Total quality management (TQM):* This collection of methods is oriented towards improving the quality culture of the organization including helping to define, implement, and track improvement goals.

- *Defect prevention process (DPP):* This method, pioneered at IBM in the 1980s, assists in categorizing defects such that they can be systematically removed and avoided in future software development products and activities.

- *Cleanroom software development:* Cleanroom is a software production method that originated in the Federal Systems Division of IBM in the late 1970s and early 1980s. Cleanroom combines practices of formal specification, nonexecution-based program development, incremental development, and independent statistical testing.

3 Estimation

3.1 Overview

Estimation is a way of determining the characteristics of a project befoıe it begins, to permit planning of resource use across the life of the project. Estimation tries to answer two important questions: how many people will the project require (at each development stage), and how long will the project take from start to finish? Several approaches have been taken to answering these and related questions; the trend in software development has been toward actuarial models, which rely on statistical techniques. Successful use of these techniques will allow more efficient allocation of project resources, by assuring that resources are available when needed, and minimizing instances of resources sitting idle. Estimation is a collection of methods leading to better project planning.

3.2 History/Background

The practice of estimation is very old; its applications to construction projects predate the invention of the computer. Statistical methods of estimation have arisen primarily in this century, when the mathematics required have become widely available. The invention of the computer and the microcomputer have aided in the practice of actuarial estimation since computers permit automation of the often tedious calculations required. Perhaps surprisingly for a method so old, estimation remains an area as troubled as it is necessary, due largely to inherent features of the estimation problem. As a method for improvement, estimation has potential benefits if it is used carefully and in a sophisticated manner.

3.3 Estimation Models

There are many estimation models. Commonly applied models include COCOMO [Boehm 81], SLIM [Putnam 92], and PRICE-S. While they are all different (in their choices of ways of counting, underlying assumptions, etc.), all are conceptually similar. We will not, therefore, describe a particular model; instead, we will focus on the basic notions embedded in all models.

As usually conceived, the basic estimation problem consists of the statistical problem of relating independent variables like "system size" (which may be measured in a several ways) to the dependent variable "staff-months of development effort." When independent variables are filled in with values for a specific project, estimation produces as output (1) a prediction of the expected number of staff-months of effort required for that project, and (2) an error term that quantifies the expected magnitude of error in the prediction. Generation of predictions and error terms requires construction of a statistical model.

3.3.1 Statistical Models

A statistical model is characterized by (1) the form of the mathematical equation chosen (i.e., linear, $y = ax + b$; exponential, $y = ax^b$, etc.); and (2) the parameters of the equation (i.e., a and b in the model equations). Form and parameters are chosen by fitting the model to available data, using statistical techniques. For example, suppose you have a database of 200 projects within your organization. Suppose further that you know the size (measured, say, in thousands of lines of source code) for each delivered system, and the staff-months required for completion of each corresponding project. You would then choose a form of equation and use statistical fitting techniques to determine the values of the parameters, so that the model related size to staff-months across all 200 projects as closely as possible. By repeating the process for different model forms, you would eventually discover a form and set of parameters that provided the "best fit" – that is, which seemed to best describe the "shape" of the data. (Statistical techniques are available to determine "goodness of fit".) You would then have a model to tell you how many staff-months of effort were required for any project for which you knew the size. The model would also supply an estimate of the error in prediction.

In practice, several difficulties beset development of such a model. Because the error term gets smaller as the number of projects grows larger, development of a model requires a database as large as possible. Also, creation of an estimation model requires statistical sophistication. For these reasons, many organizations that use estimation adopt standard estimation models rather than develop their own. The standard models make use of model forms chosen to provide good fits over a wide variety of application domains. When the adopting organization has a sufficient database, the model parameters may be customized (i.e., "calibrated") to the particular organization in which the model will be used [Boehm 81]. Developing such a database can take a number of years, dependent on the frequency and duration of new projects.

3.3.2 Staff-Month Estimates

The usefulness of staff-month estimates should be clear. Estimation models always provide conversion equations and factors, which can be used to convert staff-month estimates into estimates of project duration, staffing levels required at each stage in the life of a project, etc. Multiplication of staffing levels times labor rates provides cost estimates. The models usually also provide ways of calculating the additional staffing required to reduce project duration by a certain amount. In practice, all of these calculations are usually automated.

3.3.3 Function Point Analysis

One approach which has proven to help reduce the error involved with software size estimation is function point analysis (FPA) [Albrecht 79], [Albrecht 83]. Function points are calculated by examining functional characteristics of the software system in terms of inputs, outputs, interfaces, files, and complexity. Function points have value for comparing projects using different programming languages. They can also be calculated from a requirements or design specification prior to the start of coding. Thus more accurate estimates can often be made at earlier phases of the software development process using FPA. Jones [Jones 91] has extended FPA application to real-time systems, called feature points.

3.3.4 Independent Variables

One problem with software size estimation is that software development is a very complex process, with a great number of independent relevant variables. One way of reducing the size of the error term is to incorporate more independent variables – things like measures of where the system resides on a scale from very ordinary types of applications, like accounting systems, to completely unprecedented applications, like missile defense systems. It is not difficult to think of a great number of independent variables that affect the staffing requirements and duration of a project: complexity of application, number of contractors involved, sophistication of personnel, whether the client is a private firm or government agency, etc. Mohanty, writing in 1981, lists 49 different independent variables, each one of which is included in at least one commercially available estimation model.

But while adding a great many independent variables works in one sense to reduce the size of the error term, it also leads to another problem. In order to make good predictions about a particular sort of system, a statistical model requires data on similar systems. The greater the number of similar systems and the more similar the systems are, the more reliable predictions will be. It is not hard to see, then, that adding a great many independent variables would require much more data in order to fit a good estimation model. As more independent variables are added, the database is divided into more and more groups, and systems get distributed across these increasingly numerous groups. Take a database of 200 systems classified only by size into three categories (say, small, medium, large); there are, then, an average of 67 systems in each group. If we divide further into two additional categories (say, government client, non-government client), we now have six groups with an average of 33 systems in each group; and so on. This problem presents substantial statistical challenges. One mitigating factor is that systems tend to "cluster" in groups; for example, there tend to be a lot of very similar private branch exchange (PBX) systems. Therefore, intuitively enough, statistical models are best able to predict for types of systems for which there is much available data. Unfortunately, those are usually the systems we need help with least.

3.3.5 Expert Judgment Based Estimation

Expert judgment based estimation is an alternative or supplement to estimation models. In expert based judgment, an expert compares projects in his or her experience for similarities and differences, then makes a subjective estimate for the project in question. While this method is sometimes used by itself with some success, it is not very satisfactory in general. Experts with a broad enough range of experience to be able to estimate accurately are rare; thus, the method often becomes quite ad hoc, frequently employing nonexperts [Adrangi 87].

3.4 Difficulties with Estimation Models

Despite the conceptual simplicity of estimation models, several difficulties impede their use. To a large extent, the difficulties arise from the inherent nature of the estimation problem – as someone wise once said, "accurate prediction is difficult, especially of the future." Difficulties fall into roughly three categories: (1) challenges of obtaining good independent variable values, (2) balancing of the number of independent variables against scarcity of data, and (3) accounting for the "human element" in estimation. We consider these difficulties below.

The estimation models discussed earlier required as input a number that represents the size of the system. But the size number may itself be difficult to obtain. This fact would seem to call for the creation of "sizing models" that would take functional requirements of the system as inputs and produce as output an estimate of the size of a system that would meet the requirements. There are such sizing models, but they are typically quite primitive [Boehm 81]. The fundamental difficulty in converting information available earlier than size into effort estimates is a sort of "Catch-22" of estimation: estimates are most valuable when provided early, but they are least reliable when provided early. Many estimation models are capable of accepting independent variables that are available early, much earlier than size; but, precisely because these independent variables are available so early, they are only weakly related to project duration and staffing levels. A "garbage-in, garbage-out" situation occurs. Estimates made very early in the development process will often result in larger errors. As Case [Case 86] observes, estimation models "do a fairly good job of telling you how long the project will take – after you have written the code and then counted the lines." There is progress being made toward solving this problem [Mukhopadhyay 92], but it is still in the research stage.

One final category of difficulties with estimation models might be said to be due to "human factors." During the course of a project, project managers are well aware of estimates and may take estimates into account in choosing their actions [Austin 93], [Abdel-Hamid 86]. To some extent, this behavior may be desirable. A manager falling behind schedule may take steps to get back on schedule, and this can be good. However, some actions that managers may take in reaction to estimates may not be desirable. For example, managers or developers may be overly optimistic, which may result in the "90 percent syndrome" wherein they claim to be 90 percent done for the last 50 percent of the project [Boehm 81], [Abdel-Hamid 88]. In extreme cases, managers may actually willingly compromise the quality of a project to stay consistent with estimates [Austin 93]. Another problem that may arise, when schedules are too generous, is that managers and developers may expand the amount of work they do and the functionality in the system to fit the schedule, when the client would rather have the system earlier [Abdel-Hamid 86]. Statistical estimation models rarely explicitly account for the human element.

3.5 Experience with Estimation

Every organization that develops software does some sort of estimation. The planning necessary to bring resources to bear on a task requires at least some rudimentary level of estimation. This estimation can be either ad hoc (e.g., "how long do you think it'll take?" "I dunno, how long do you think it'll take?"), or systematized in some way. Experiences with ad hoc methods have lead many to argue that there must be some advantage in systemization. DeMarco [DeMarco 82] argues that ad hoc methods do not adequately support learning based on estimation experience and, furthermore, that they have been generally ineffective. The question, then, is whether actuarial estimation models can provide some advantage over ad hoc methods.

This is not a very severe test – ad hoc methods are so widely regarded as unacceptable, that even a small improvement would be beneficial. However, there is a legitimate question as to whether use of actuarial models provides improvement. Mohanty [Mohanty 81] claims that "almost no model can estimate the true cost of software with any degree of accuracy." A U.S. Air Force Avionics Laboratory study in 1984 concluded that none of the available models at that time could be shown to be accurate enough to justify their use [Ferens 88]. Kusters et al [Kusters 90] recommend never using estimating models by themselves, because they are not accurate enough. Van Genuchten and Koolen [van Genuchten 91] claim that "no studies confirm the accuracy and usability of the (currently available) models."

Mohanty [Mohanty 81] carried out an experiment in which the estimates produced by 12 different models were compared for the same system. Estimates from these models ranged dramatically; the highest estimate was roughly 700 percent greater than the lowest. Many have reasoned that the problem lies in the specific databases used by each model; there are simply too many organization-specific independent variables still buried in the data [Mohanty 81], [Clapp 76], [Bartol 82], [Tausworthe 77]. One implication of this conjecture, if true, is that estimation models fitted with data from applications similar to applications in the environment where the tool will be used will be more accurate predictors in that environment. Ferens [Ferens 88] has observed that, for this reason, a particular tool may be best for a particular environment. A study by Navlakha [Navlakha 90] suggests that organizations often use the wrong model for their environment. Another problem with the databases used to fit model parameters is that they are almost always old, relative to the rapid advance in complexity of software applications [van Genuchten 91].

The portability of models, i.e. their validity when transferred from one organization to another, has also been determined to be quite poor [Benbasat 80]. This is a serious matter in organizations that lack their own large database of projects to use in calibrating a model to their own use. Kemerer [Kemerer 87] concludes in an empirical study of four of the most popular models that there is considerable need to customize the model to the application environment; without calibration, error rates were often in the 500 to 600 percent range. He also notes that "all of the models tested failed to sufficiently reflect the underlying factors affecting productivity." Zelkowitz et al. [Zelkowitz 84] report that organizations distrust estimating models and often use them only to compare against manually generated estimates. Van Genuchten and Koolen (1991) cite a European survey by Siskens et al. [Siskens 89] that finds only 14 percent of respondents using estimation models; van Genuchten and Koolen argue based on their own observations that even those who claim to use estimation models often do not use them successfully.

Not all of the news is bad, however. Kemerer found that when calibration was done, the best of the models were able to account for 88 percent of the actual staff-month effort (that is, only 12 percent of variation was in the error term). Navlakha [Navlakha 90] found an instance where a particular model predicted for a particular environment with only 8.3 percent error. This is certainly an improvement over ad hoc methods. As Kemerer also notes, however, some of the benefit of using estimating models may come from the degree of structure they impose on the estimating task. Even when developers do not evaluate estimation models as "good," they do concede that they are "useful" [van Genuchten 91]. There are also reports of promising results in organizations that are beginning to use estimating models. Lehder et al. [Lehder 88] report that within AT&T use of estimating models has, on pilot efforts, resulted in estimates within 20 percent of actual experience – again, clearly an improvement over ad hoc methods.

Although it is difficult to quantify, perhaps the biggest impact of estimation application is its use as a training and communication vehicle. Estimation is fundamental to the planning of software projects. When software project managers initially learn estimation methods they are also learning project planning skills. In addition, the variables to be estimated such as size, staffing levels, and schedule, become the measures that project managers and their managers will track during the course of the project. Thus, we believe that there is a correlation between better project planning and estimation.

3.6 Suggestions for Introduction and Use

The literature suggests that estimation models should be used with great caution. Blindly accepting estimates is, by all accounts, a recipe for disaster.

It seems clear from a number of sources (e.g., [Kemerer 87], [Adrangi 87], [Ferens 88]) that the models are often of little use without calibration. This would suggest that organizations that have available a database of their own projects to be used in calibration are in the best position to benefit from an estimation model. Furthermore, some models are better suited for certain environments than others [Ferens 88], [Kemerer 87]. Therefore, the potential for benefit will be maximized if the adopting organization first conducts a comparative study, to determine which model is best suited to the organization [Navlakha 90].

There is some evidence that the additional structure imposed on the estimating task by an estimating model is beneficial in itself [Kemerer 87], [van Genuchten 91], [Lehder 88]. If so, then it is perhaps worthwhile to use estimating models in conjunction with other modes of estimating (such as the expert judgment method discussed briefly above). Several authors make precisely this recommendation [van Genuchten 91], [Kusters 90].

Van Genuchten and Koolen [van Genuchten 91] suggest the following process:

1. Use an expert estimator to obtain a subjective estimate.
2. Use an estimating model to obtain an estimate.
3. If the estimates agree, accept the estimate; if not, the reasons for differences must be determined.
4. Through the process of exploring differences, come to a consensus estimate.

Van Genuchten and Koolen [van Genuchten 91] also list the following organizational requirements for successful use of an estimating model:

- Cooperation of software developers.
- Availability of staff to introduce and use the model.
- Commitment of management.
- Estimation guidelines to assure proper use of estimation methods.
- Adequate information supply to support use of the model (e.g., data gathering on projects to support refinement of the model's parameters).

3.7 How Estimation Is Related to the Capability Maturity Model

Estimation is a skill required for successful project planning. Within the Capability Maturity Model, estimation is a software process improvement method which primarily correlates to the level 2 key process area, software project planning.

3.8 Summary Comments on Estimation

Most organizations feel that they need help with estimation. Estimation models provide some help, but there is danger in placing too much hope in them. However, if comparative testing is done to determine the best model for the organization in question, and if calibration data are available in the adopting organization, the estimation model can be helpful. When estimation models are useful, it is often in conjunction with other estimation techniques. Thus, we recommend that estimation models be employed in favorable circumstances, and in conjunction with other methods.

3.9 References and Further Readings - Estimation

[Abdel-Hamid 86]Abdel-Hamid, T. K., Madnick, S. E., "Impact of Schedule Estimation on Software Project Behavior," *IEEE Software*, 70-75, Vol. 3., No. 4, July, 1986.

[Abdel-Hamid 87]Abdel-Hamid, T. K., Madnick, S. E., "On the Portability of Quantitative Software Estimation Models," *Information and Management*, 13, 1-10, 1987.

[Abdel-Hamid 88]Abdel-Hamid, T. K., "Understanding the '90% Syndrome' in Software Project Management: A Simulation-Based Case Study," *Journal of Systems and Software*, Vol. 8, No. 4, 319-330, September, 1988.

[Adrangi 87]	Adrangi, B. and Harrison, W., "Effort Estimation in a System Development Project," *Journal of Systems Management*, 21-23, August, 1987.
[Albrecht 79]	Albrecht, A.J., "Measuring Application Development Productivity," *Proc. of the Joint SHARE/GUIDE Symposium*, pp. 83-92, 1979.
[Albrecht 83]	Albrecht, A.J. and Gaffney, J.E., "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 6, pp. 639-648, 1983.
[Austin 93]	Austin, R.D., "Strategic Response to Time Pressure During Systems Acquisition," Working Paper, Dept. of Social and Decision Sciences, Carnegie Mellon University, 1993.
[Bartol 82]	Bartol, K. M., Martin, D. C., "Managing Information Systems Personnel: A Review of the Literature and Managerial Implications," *MIS Quarterly*, 49-70, December, 1982.
[Benbasat 80]	Benbasat, I., Vessey, I., "Programmer and Analyst Time/Cost Estimation," *MIS Quarterly*, 31-44, June, 1980.
[Boehm 81]	Boehm, B. *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
[Case 86]	Case, A. F. *Information System Development*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
[DeMarco 82]	DeMarco, T., *Controlling Software Projects*, Yourdon Press, New York, 1982.
[Ferens 88]	Ferens, D. V. "Software Parametric Cost Estimation: Wave of the Future," *Engineering Costs and Production Economics*, Vol. 14, 157-164, 1988.
[Jones 91]	Jones, C., *Applied Software Measurement*, McGraw-Hill, New York, 1991.

[Kemerer 87] Kemerer, C. F., "An Empirical Validation of Software Cost Estimation Models," *Communications of the ACM*, Vol. 30, No. 5, 416-429, May, 1987.

[Kusters 90] Kusters, R. J., van Genuchten, M., Heemstra, F. J., "Are Software Cost-Estimation Model's Accurate?," *Information and Software Technology*, Vol. 32, No. 3, 187-190, April, 1990.

[Lehder 88] Lehder, W. E., Smith, D. P., Weider, D. Y., "Software Estimation Technology," *AT&T Technical Journal*, July/August, 10-18, 1988.

[Meyers 78] Meyers, W., "A Statistical Approach to Scheduling Software Development," *IEEE Computer*, 23-35, 1978.

[Mohanty 81] Mohanty, S. N., "Software Cost Estimation: Present and Future," *Software - Practice and Experience*, 103-121, 1981.

[Mulhopadhyay 92] Mulhopadhyay, T., Kekre, S., "Software Effort Models for Early Estimation of Process Control Applications," *IEEE Transactions on Software Engineering*, Vol. 18, No. 10, 915-924, October, 1992.

[Navlakha 90] Navlakha, J. K., "Choosing a Software Cost Estimation Model for Your Organization: A Case Study," *Information and Management*, Vol. 18, 255-261, 1990.

[Putnam 92] Putnam, L., Meyers, W., *Measures for Excellence: Building Reliable Software on Time, Within Budget*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[Siskens 89] Siskens, W. J., Heemstra, F. J., van der Stelt, H., "Cost Control in Automation Projects: A Survey," *Informatie*, Vol. 31, January, 1989 (in Dutch).

[Tausworthe 77] Tausworthe, R. C., *Standardized Development of Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[van Genuchten 91] van Genuchten, M., Koolen, H., "On the Use of Software Cost Models," *Information and Management,* Vol. 21, 37-44, 1991.

[Zelkowitz 84] Zelkowitz, M. V. "Software Engineering Practices in the US and Japan," *Computer,* 57-66, June, 1984.

4 ISO 9000 Certification

4.1 Overview

ISO 9000 is a series of quality standards established in 1987 by the International Standards Organization (ISO). Organizations are assessed by independent third parties to determine whether their business practices meet one of the standards; an organization that meets the appropriate standard can legally advertise its "ISO 9000 certification." Certification is increasingly important for companies that trade internationally. All firms that operate within the European common market are required to be ISO 9000 certified, and the standards have been adopted in greater or lesser degree by 35 countries around the world [Marquart 92]. Even in countries where certification is not mandated, customers are increasingly asking of prospective suppliers, "are you certified?" The purpose of the ISO 9000 series of standards is to provide a means by which customers can be assured that their suppliers are using certain quality business practices without performing individual audits of each prospective supplier.

4.2 History/Background

The ISO 9000 series of standards was written by a committee of the ISO composed of delegates from many nations. Delegates created the standards by combining aspects of military standards, nuclear power plant regulations, medical device manufacturer regulations, and other regulations and standards from delegate countries. ISO 9000 is widely deployed in the European Community (EC), as well as in countries with close ties to the EC (e.g., New Zealand). U.S. manufacturers like DuPont, Union Carbide, and a variety of others, who trade in significant volume with EC clients, are busily certifying their facilities to maintain access to their markets. The standards were conceived in the context of manufacturing operations and certification is at the facility level. Benefits of certification are widely espoused by companies that have been certified, but there is no solid empirical evidence that certification is beneficial. Because certification is fairly new, long-term effects are not well known. This may, however, be a moot point for companies who find their clients demanding ISO 9000 certification as a condition of continuing business.

4.3 Description of ISO 9000

The ISO 9000 standards establish requirements for the systems of production within the facilities of a company. They are not product standards, but rather process standards, which require that certain process characteristics are in place. For example, one standard (ISO 9002) contains stipulations such as "those processes affecting quality must be monitored and controlled" and "objective evidence must be provided that the product received and delivered is inspected or otherwise verified" [Gasko 92].

There are five primary standards, numbered 9000 through 9004. ISO 9000 provides basic definitions and concepts and describes how the other standards in the series should be used. ISO 9001, 9002, and 9003 are the actual standards against which companies are certified. When a company states that it is ISO certified, it most often means that it has met ISO 9001 or 9002.

- ISO 9001 is for facilities engaged in design, production, installation and servicing of a product.
- ISO 9002 is a less detailed standard for facilities engaged only in production and installation.
- ISO 9003 is an even less detailed standard for organizations that seek only to certify their conformance to accepted practices in final testing and inspection of products.

Unlike 9001 through 9003, ISO 9004 is not intended to be used to establish guarantees of practices within the organization that uses the standard. The 9004 standard is, rather, for internal use only and lists the elements of a comprehensive total quality system. It provides guidelines with which a facility can evaluate its practices in marketing, design, procurement, production, measurement, post production, materials control, documentation, safety, and use of statistical methods.

The ISO 9000-3 standard [ISO 90] provides guidelines to organizations that produce software products. The ISO 9000-3 provides additional guidance information to software development organizations such that they can apply the more general ISO 9001 standard. It thus helps correlate software process terminology to the quality systems requirements identified in ISO 9001. The ISO 9000-3 standard identifies some of the software process practices as described within the Capability Maturity Model such as testing, configuration management, change control, measurement, training, and software quality assurance.

The standard approach to achieving ISO 9000 certification involves comprehensively documenting all processes involved in production and in support of production. In the course of documenting, processes are reviewed and evaluated to determine if they comply with the standard. When they do not, corrective actions are taken, and new processes are created. The eventual goal is to create a situation in which "you document what you do, and do what you document" [Marquardt 92]. Successfully achieving this situation and standard levels of quality of product will result in certification. Usually informal self-assessments are made at various stages on the way to certification, to determine the likely outcome of a formal assessment. When results of self-assessments are satisfactory, the organization employs a third party to conduct a certification assessment.

4.4 Experience with ISO 9000

ISO 9000 is a relatively new business development. We know of no rigorous empirical studies of its effectiveness. The long-term effects of certification are not known. Supporters point out that the standards simply require businesses to conform to practices that are widely considered effective. We note, however, that "widely considered effective" may not be the same thing as "effective." The issue of how mandating practices transforms the nature of what is being required in the minds of those to whom the requirement applies also needs to be addressed.

Inevitably, in the early stages of a business movement, the literature is full of claims of benefits by those who have vested interests in promoting the movement. Most early evidence comes from individuals in companies who were responsible for instituting a program and are now claiming credit for its success, or from consultants who stand to gain business from heightened interest in the movement in question. Many reports of experience with ISO 9000 have this flavor. Eventually movements either prove to have real benefits and become part of the way of doing business, or they disappoint or turn out to have undesirable long-term side effects. The impact of ISO 9000 on software process improvement remains to be seen.

4.4.1 Reported Benefits

There are claims being made about the benefits of ISO 9000. DuPont reports benefits such as increased manufacturing yields, decreased customer complaints, reduction in process deviations [Dzus 91], increases in on-time delivery, decreases in cycle time, increased first pass yields in production lines, and more [Marquardt 92]. ICI Advanced Materials claims reduced product costs due to less rework and fewer customer returns [DeAngelis 91]. The British Standards Institution, a leading certification assessor, estimates that registered firms reduce operating costs by 10 percent on average [Marquardt 92]. In short, the benefits of ISO 9000 seem to be the benefits of any good quality program.

In addition to promoting quality practices, the ISO 9000 series of standards has a clear coordinative purpose. As products become more complex, it is increasingly difficult to ensure the quality of materials and components provided by suppliers. One response to this difficulty has been for client organizations to take an interest in the production processes of their suppliers. But it is expensive to audit every supplier's processes. Certification is a means of establishing immediately that the supplier's processes have certain desired characteristics. In this sense, the ISO 9000 certification is not unlike, say, U.S. Department of Agriculture certifications on farm produce that assure a certain quality thereby saving buyers from needing to perform expensive tests on their incoming product. It is, however, worth following up this analogy. Recently, farm produce standards have come under fire for relying too much on appearance and, consequently, for causing farmers to use far too much pesticide [Austin 93]. One wonders what similar destructive bias might eventually prove to be inherent in the ISO 9000 standards.

It has also been observed that standards have a generally destructive side, and that they are difficult to dislodge or modify once they are in place. Ishikawa [Ishikawa 85] notes that "even when industrial standards are modified, they cannot keep pace to customer requirements." There is both an upside and downside to proceduralization. The benefit is that the procedural knowledge of the business does not reside exclusively within personnel, which, among other things, moderates the effects of worker turnover. The organizational memory resides in the procedures, as well as in the people. However, procedures are inherently inferior to people in adapting to the unusual circumstances that inevitably arise in a production environment. The downside of proceduralization, then, is that attention to procedures may obscure the view of what really needs to be done. ISO 9000 requires much documentation, which sometimes can be associated with bureaucracy and unresponsiveness.

4.4.2 Potential Problems

One may question the effect of mandating a standard as a condition of doing business. Obviously a supplier could institute quality programs without being required to. What impact then does the requirement have? There are some precedents that allow some conjecture on this matter.

Requiring that all participants in a market meet the same high standard of process and product quality may ensure that certain practices are commonly practiced across an industry. But is it necessarily a good thing to have most of the companies in an industry doing things in the same ways? Supporters may protest that it is not the purpose of the ISO 9000 standards to induce homogeneity of process across all companies; there are, they might say, many ways to quality and the ISO 9000 standards are capable of accommodating wide variations in methods. Such defenses, though, ignore the nature of the human response to situations where requirements are linked with very great rewards or severe punishments. A company whose access to markets, whose very business future depends on certification, is likely to use the most widely accepted methods, to take the safest course to ensure certification. This will inevitably result in a certain degree of homogeneity, as a small subset of "good" consultants establish their success rates in achieving certification for their clients, by promoting an ever shrinking set of techniques. In the worse case, process innovators are punished for not complying with industry standards.

Also, when standards are required, a subtle sort of goal displacement takes place in the minds of those working toward the goal [Merton 57]. The goal becomes winning the certification, not achieving the benefits of the program. Motivation becomes extrinsic, oriented toward an evaluative body which will pass judgment. Large sums begin to be invested in preparation for the audits. Those wishing to be certified grow cleverer and consultants grow richer. In the U.S., some of the same consultants that train organizations in ISO 9000 are also certification auditors [Fouhy 92]. Fouhy et al. [Fouhy 92] quote one consultant who describes how certification for its own sake overtakes any quality goals a company might have and summarizes by saying, "currently 80-90% of the companies going through the certification process are wasting their money." A representative from industry states that if certification is not part of an overall strategy, it becomes just "extra paperwork."

4.5 Suggestions for Introduction and Use

For companies whose customers demand ISO 9000 certification, there is little benefit in debating its merits. These companies need to know how to become certified. Not surprisingly, this is the focus of much of the literature on ISO 9000.

DeAngelis [DeAngelis 91] provides the following list of "dos and don'ts" based on ICI Advanced Materials' successful certification effort:

1. Do get top management support.
2. Do allow ISO 9000 to complement existing quality programs. Conduct quality sessions as well as ISO 9000 seminars to ensure that people understand the process goals. Involve all staff levels in the seminars.
3. Do survey the seminar attendees to ensure that the program addresses their needs. Seminars should reflect the distinct needs of each group.
4. Don't have outsiders write procedures. The procedures should be written by those most familiar with each job.
5. Do solicit on-the-job volunteers to draft the ISO 9000 job procedures.
6. Don't make volunteers operate in a vacuum. Provide format, support, and deadlines.
7. Do review drafts with all involved in the procedure to ensure support.

8. Do provide incentives and recognition for ISO 9000 volunteers.

9. Do communicate progress to employees regularly to show them how ISO 9000 has affected performance and sales.

10. Don't expect ISO 9000 to solve quality problems. Recognize that it is a baseline for good business practice, and that direct continuous improvement efforts are required to build on the ISO 9000 foundation.

Similarly, Dzus [Dzus 92] documents the 20 steps DuPont followed to certification in their Towanda, Pennsylvania facility:

1. Obtained management commitment and began training for leaders and coordinators.
2. Set up a steering committee and sub-teams to identify what needed to be done.
3. Began internal quality auditing.
4. Held a two-day, in house seminar.
5. Compiled a quality manual.
6. Conducted ongoing quality audits and implemented corrective actions.
7. Filled out the application for assessment and paid the application fee.
8. Submitted the quality manual to the auditor for assessment.
9. The pre-assessment was conducted.
10. Implemented improvements based on pre-assessment recommendations.
11. Set the date for being ready for assessment.
12. Set the assessment date for two to three months later.
13. Provided more in-house training.
14. Conducted frequent checks on status with plant leaders.
15. Got plant leaders to conduct audits.
16. Shared pre-assessment findings.
17. Sent revised quality manual to auditor.

18. The assessment was conducted.

19. Fixed minor discrepancies.

20. Registration obtained.

Gasko [Gasko 92] provides the following account of the certification experience of Union Carbide's Taft Louisiana plant:

1. Set up a steering committee.
2. Conducted a "gap analysis" to determine where business processes were not up to standard.
3. Established a three-tiered document system composed of (i) a company quality system manual that makes a broad policy statement, (ii) a facilities manual that specified what needed to be done and by whom, and (iii) work instructions that told how to do what needed to be done.
4. Wrote these documents in parallel, maintaining close contact between people responsible for writing each.
5. Adapted procedures already in place to fit with ISO 9000 standards. Sometimes this merely required changing the names of procedures.
6. Allowed individuals to work on their own in developing procedures. Management focused on cutting red tape.
7. Set an aggressive time frame for certification.
8. Avoided bureaucracy and emphasized individual accountability.

The preparation for certification can take about one to two years [Dzus 92], [Fouhy 92], [DeAngelis 91].

4.6 How ISO 9000 Is Related to the Capability Maturity Model

Since ISO 9000-3 is primarily guidelines for developing software quality management and quality assurance standards, it most closely correlates to the level 2 key process area (KPA), software quality assurance. It also minimally refers to some other KPAs such as software subcontract management, software project planning, software project tracking and oversight, peer reviews, software product engineering, configuration management, organization process definition, and training. Although the ISO 9000-3 guidelines do not contain as much description of practices as the CMM, we can roughly correlate the maturity level of a software organization that has ISO 9000 certification to at least level 2, repeatable. The ISO 9000 certification audit process encourages the development of software standards and procedures documents that correspond to the ISO 9000-3 requirements. What ISO 9000 certification does not directly address is the question of whether or not the documented practices are effective. Thus, there exists the possibility that the ISO 9000 certification could be given to an organization that documents and implements poor practices or does not consistently follow its documented practices.

4.7 Summary Comments on ISO 9000

While we have few qualms with the quality philosophies and methods espoused by the ISO 9000 standards, we feel compelled to note the potentially detrimental effects of mandatory standards. We note that Japanese companies have been unenthusiastic about third party certification of quality systems [McFadyen 92], because, we conjecture, their own quality systems are based on intrinsic desires to do better, and not an extrinsic need to be certified or win an award. As noted already, however, such reservations are irrelevant if your customers are demanding that you be certified. A big advantage of the ISO 9000 method is that it becomes a strong motivator for paying attention to software process improvement when certification is necessary to remain competitive.

4.8 References and Further Readings - ISO 9000

[Austin 93] Austin, R. D., "A Theory of Measurement and Dysfunction in Social Institutions", Ph.D. Dissertation, Pittsburgh, Pa.: Carnegie Mellon University, 1993.

[DeAngelis 91] DeAngelis, C. A., "ICI Advanced Materials Implements ISO 9000 Program." *Quality Progress*, November, 1991.

[Dzus 91] Dzus, G., "Planning a Successful ISO 9000 Assessment." *Quality Progress*, November, 1991.

[Fouhy 92] Fouhy, K., Samdam, G., Moore, S., "ISO 9000: A New Road To Quality." *Chemical Engineering*, October, 1992.

[Gasko 92] Gasko, H. M., "You Can Earn ISO 9002 Approval in Less Than a Year." *Journal for Quality and Participation*, March, 1992.

[Ishikawa 85] Ishikawa, K., *What is Total Quality Control? The Japanese Way*. Translated by David J. Lu, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[ISO 90] ISO/DIS 9000-3, *Quality Management and Quality Assurance Standards - Part 3: Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software*, Sept. 1990.

[Marquardt 92] Marquardt, D. W., "ISO 9000: A Universal Standard of Quality." *Management Review*, January, 1992.

[McFadyen 92] McFadyen, T., Walsh, T., "Is ISO 9000 Worth the Paper It's Written On?" *Journal for Quality and Participation*, March, 1992.

[Merton 57] Merton, R.K., *Social Theory and Social Structure*, Free Press, Glencoe, IL, 1957, p. 50.

5 Software Process Assessment (SPA)

5.1 Overview

Software process assessment (SPA) is a means of determining the strengths and weaknesses of an organization's software development process. Assessments are done by trained members of the organization being assessed or by assessment vendors. Results of assessments are confidential. Results include a "maturity rating" from 1 to 5 (1 is lowest, 5 is highest) and a detailed account of the strengths and weaknesses of the organization's development process. One purpose of an initial assessment is to form a baseline from which an organizational strategy for process improvement can be constructed. The initial assessment is often an organizational intervention to help obtain "buy-in" for change. Reassessments are means by which progress can be measured.

5.2 History/Background

Software process assessment is a relatively new technique, even by the short-term standards of the software industry. The method was derived largely from the work of Watts Humphrey [Humphrey 89] and his colleagues at the Software Engineering Institute (SEI) in the late 1980s, inspired by Crosby's [Crosby 79] work on quality assurance. Since the SEI is a federally funded research and development center with commensurate ability to influence U.S. government policy, assessment has been widely deployed in a short period of time. One reason for such rapid deployment is the close association between assessments and software capability evaluations (SCEs), an evaluation of development organizations intended to determine their suitability for government contract work. Assessments are often done in preparation for SCEs, because the methods are based on the same model of a "mature" software development organization.

Despite their widespread use, neither assessment nor the underlying maturity model (which also underlies SCEs) has been conclusively demonstrated to improve software development results. Empirical validation of the model is beginning.

5.3 Description of Software Process Assessment

Assessments (and SCEs) make use of the SEI's Capability Maturity Model (CMM) [Paulk 93]. The maturity model depicts stages of evolution of a software development organization from immature to most mature. The questionnaires that begin the assessment process, and the interviews and discussions that follow, are aimed at determining how current practice in the organization compares with the CMM ideal.

Assessment begins when high-level managers of an organization decide that they want their organization assessed and commit to providing the necessary level of resources. Because it starts at the top, proponents argue, assessment is more likely to result in actual implementation of improvements. Also, participants in the assessment, seeing the support of management, will take the process more seriously and feel more confident that their input might cause a change in the organization [Bollinger 91].

Next, a team from the organization to be assessed is trained in performing assessments by the SEI itself, or by trainers accredited by the SEI. Alternatively, a team from an SEI-licensed vendor or from the SEI itself can be engaged to perform or help perform the assessment. However it is accomplished, great care is taken to protect the confidentiality of the results.

The assessment team selects certain projects and groups within the organization for assessment interviews, and representatives from these projects and groups are asked to complete a questionnaire about their software development process. The questionnaire serves as a starting point for an intensive week of interviews and group discussion that follows shortly thereafter.

The intensive three- to five-day period of interview sessions and guided group discussions are what is usually meant by "assessment" in common usage. Assessment team members interview people about how they responded to the questionnaire and about issues that arise in discussion, all under strict guidelines concerning confidentiality and nonattribution of information to sources. Because of the extreme emphasis on confidentiality, these discussions are often characterized by open, free exchange of candid beliefs about the current state of the organization's processes. Participants in the assessment are asked to reach consensus on how they would improve the software development process, and on the last day of the assessment, key findings and recommendations are presented to management.

After the week of interviews and discussions, the assessment team completes the process by writing a report containing their key findings and recommended action plan for achieving improvement. This report is conveyed to management so that recommendations can be implemented. Management follows through by acting on the recommendations in the report. Reassessment is recommended in 18-24 months.

5.4 Assessment Results

According to a recent SEI study [Kitson 92], most assessed organizations are at the initial maturity level. Only 12 percent were discovered to be at the repeatable level, and 7 percent at the defined level. There were no organizations identified in the study at the managed or optimizing level. Only one project in the United States is believed to operate at the optimizing level, the project supporting the fly-by-wire software for the space shuttle [Keller 93].

5.5 Other Assessment Methods

Although this section primarily describes the assessment approach developed at the SEI, it is important to note that it is only one of a number of existing assessment methods. For example, an ESPRIT project, Bootstrap, has used the CMM, but it has extended the SEI method to better apply to the European software industry [Card 93]. The Bootstrap method has augmented the SEI method to better reflect the requirements of the ISO 9000 quality standards. In addition, a number of software consulting firms have developed proprietary assessment methods to support their customers. Many larger corporations (e.g. Motorola, AT&T, IBM, Siemens) have also modified/enhanced the SEI method to better fit within their corporate culture. In late 1992, an ISO Working Group was established to address the issue of standardization of assessment methods. This effort is popularly known as the SPICE project (Software Process Improvement and Capability dEtermination).

5.6 Experience with Software Process Assessment

Assessment is one of the newest of the methods described in this report. For this reason, there is relatively little published evidence of the method's success. Some case study data are available [Dion 90], [Humphrey 91], which provides reason to believe the process can be successful. Humphrey, Snyder, and Willis, for example, report that a division of Hughes Aircraft believes it spent on the order of $450,000 on assessment and subsequent improvements to gain annual savings estimated at $2 million.

The newness of the method is but one reason benefits are difficult to determine. Another is the problem that naturally arises when a concerted effort to promote practice of a method is carried out in parallel with attempts to determine effectiveness of the method. In such circumstances, it is in the interests of promoters to seek out evidence of success, but the incentive is not so great to discover difficulties. Also, there is a natural confounding of early results with promotional attempts.

On the other hand, the CMM does not present a model of software development that is new or controversial, at least not in any obvious sense. Proponents note that it is simply a coherent compilation of common notions about what software development processes should look like. It is, in the words of one advocate "refined common sense." If we are to believe in our common sense, then the CMM seems likely to be a reasonable, normative account of development processes.

5.7 Potential Problems

Assessment promoters often emphasize that the initiative and the desire for assessment should come from within the organization to be assessed. There is reason in the more general behavioral science literature to believe that this might be a critical success factor. There is a subtle but extremely important difference between intrinsic motivation and extrinsic motivation. When motivation for pursuing improvement comes from within, the spirit rather than the rule of the improvement is most important. When motivation is extrinsic – that is, if assessment is performed by higher management on an unwilling organization, in an attempt to sort villains from heroes – then the whole process may be undermined. The free and open exchange of ideas will certainly be compromised and those being assessed will set about finding ways of subverting the assessment process [Campbell 79]. A host of behavioral researchers (e.g., [MacGregor 60], [Ouchi 81]) have provided evidence that the resourcefulness of members of an organization who are intent upon defeating attempts at evaluation is essentially boundless. The psychology of motivation (see [Deci 85]) tells us also that once motivation has become extrinsic, intrinsic motivation is also compromised. In other words, once the hard feelings inevitably stirred by forced evaluation are raised, they are difficult to dispel and organization members are less likely to internalize the objectives of the organization.

It should also be noted that assessment is an initial first step for software process improvement. Once the findings and recommendations report is completed, it is necessary for the software development organization to implement the recommendations. The implementation of the recommendations requires investment, planning, tracking, and a long-term commitment to process improvement. The implementation phase is where many organizations fail to leverage their initial investment in assessment. The on-site assessment period is designed to obtain maximum buy-in and commitment to process improvement throughout the organization. Business pressures, staff turnover, strategy changes, reorganizations, etc., often negatively affect the discipline necessary to implement long-term process improvement.

5.8 Suggestions for Introduction and Use

A recommended first step for any organization wishing to begin using assessment is to contact the Software Engineering Institute in Pittsburgh, Pa. The SEI maintains lists of assessment vendors, produces technical reports on the assessment process, and provides training and expert support. Consulting the body of experience in assessment represented by these resources is a good way to find up-to-date advice on how to conduct assessments.

Humphrey, Snyder, and Willis [Humphrey 91] provide the following advice to those interested in performing assessments, based on their experience at Hughes Aircraft.

- *Management commitment.* Delegation is not strong enough to overcome roadblocks. Commitment is. Process improvement should be tied to the salary or promotion criteria of senior management.
- *Pride is the most important result.* Pride feeds on itself and leads to continuous measurable improvement.
- *A software technology center is necessary.* Combining development, project management, administration, technology development, training, and marketing in the same organization makes it more likely that the assessment process will be successful.
- *A focal point is essential.* A group commissioned to plan, coordinate, and implement organization-wide improvement efforts is a requirement.
- *Software process expertise is essential.* Improvement teams must evolve into experts in the process of software development.
- *An action plan is necessary but not sufficient.* Producing an action plan provides a basis for improvement, but there is more work to be done in exploring other things that need to be done and achieving organization-wide consensus on the need for certain actions.

Some have suggested (e.g., [Humphrey 89]) that assessments and process improvement efforts that are instigated by assessments are most successful when they are supported by a standing group assigned responsibility for promoting such efforts. Fowler and Rifkin [Fowler 90] echo Humphrey [Humphrey 89] in recommending that "software engineering process groups" (or SEPGs) be composed of full-time staff of a size that is about one to three percent of the development organization's resources. The SEPG duties include facilitating assessment efforts and supporting, tracking, and reporting on all process improvement efforts underway in the organization. The SEPG is staffed by practitioners who are familiar with the problems faced by their colleagues working on development projects. The SEPG staff is guided by a steering committee which provides management support for SEPG actions and management input into process improvement decisions.

Fowler and Rifkin [Fowler 90] report that in practice few organizations maintain standing full-time SEPG staffs at the recommended one to three percent level, but that gains can also be realized by smaller staffs working in collaboration with developers who have part-time SEPG assignments. An SEPG partially composed of active developers is more likely to remain responsive to developer concerns, but is also more likely to be "cannibalized" in times of crisis. There is a very real risk that process improvement will not take hold in organizations that abandon it every time a project deadline approaches. Organizations committed to software process improvement should consult Fowler and Rifkin (1990) which provides guidance in establishing and operating an SEPG.

5.9 How Software Process Assessment Is Related to the Capability Maturity Model

Although assessment can be applied to organizations at all levels of the Capability Maturity Model, it is most effectively applied to organizations that are focused on software process improvement and have the resources to implement the resulting action plan. Thus the assessment method best correlates to the level 3 key process area, organization process focus. Within the ideal environment, members of the software engineering process group (SEPG) would participate as members of the assessment team and then assist in implementing the improvement actions. Assessments would be repeated approximately every two years to monitor the progress of the organization in climbing the maturity level ladder, and new and modified improvement actions would be identified. In addition, measures would be defined and monitored by the SEPG to identify whether or not the improvement actions are being implemented correctly and whether they are having an impact on the organization's performance.

5.10 Summary Comments on Software Process Assessment

Assessment is unusual among the methods described in this document. It is one of the newest of the improvement methods, but it is also one of the most widely deployed and probably the best supported. It is not yet conclusively proven to be a means to improvement, but validation attempts are underway. When implemented in a manner consistent with its original intentions – that is, intrinsically motivated, largely self-assessment – it seems likely (to us) to yield benefits. The trend towards augmentation, implementation, and standardization of the SEI assessment method by an international software industry market increases the likelihood that assessment will be applied as a positive step towards process improvement.

5.11 References and Further Readings - SPA

[Bollinger 91] Bollinger, T. B., McGowan, C. "A Critical Look at Software Capability Evaluations," *IEEE Software*, July, 1991.

[Campbell 79] Campbell, D. T. "Assessing the Impact of Planned Social Change," *Evaluation and Program Planning*, Vol. 2, 1979.

[Card 93]	Card, D., Ed., "Bootstrap: Europe's Assessment Method", *IEEE Software*, May 1993, pp. 93-95.
[Dawes 91]	Dawes, R. M. *Rational Choice in an Uncertain World*, Harbrace-Court, 1991.
[Deci 85]	Deci, E. L., Ryan, R. M. *Intrinsic Motivation and Self-Determination in Human Behavior*, Plenum Press, NY, 1985.
[Fowler 90]	Fowler, P., Rifkin, S., *Software Process Engineering Group Guide*, (CMU/SEI-90-TR-24, ADA235784), Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, September, 1990.
[Humphrey 89]	Humphrey, W. S. *Managing the Software Process*, Addison-Wesley, Reading, MA, 1989.
[Humphrey 91a]	Humphrey, W. S., Curtis, B. "Comments on 'A Critical Look'," *IEEE Software*, July, 1991.
[Humphrey 91b]	Humphrey, W. S., Snyder, T. R., Willis, R. R. "Software Process Improvement at Hughes Aircraft," *IEEE Software*, July, 1991.
[Keller 93]	Keller, T., Presentation, *First International Software Metrics Symposium*, May 22, 1993, Baltimore, MD.
[Kitson 89]	Kitson, D. H., Humphrey, W. S. *The Role of Assessment in Software Process Improvement*, (CMU/SEI-89-TR-3, ADA227426), Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1989.
[Kitson 92]	Kitson, D. H., Masters, S. *An Analysis of SEI Software Process Assessment Results: 1987-1991*, (CMU/SEI-92-TR-24), Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.
[MacGregor 60]	MacGregor, D. *The Human Side of Enterprise*, McGraw-Hill, NY, 1960.
[Ouchi 81]	Ouchi, W. G. *Theory Z*, Avon Books, NY, 1981.

[Paulk 93] Paulk, Mark C. et al, *Capability Maturity Model for Software, Version 1.1*, (CMU/SEI-93-TR-24, ADA263403), Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, February 1993.

[SEI 91] Software Engineering Institute, "Process Program Releases Revised Capability Maturity Model," *Bridge*, December, 1991.

[SEI 92a] Software Engineering Institute, "SPA: SCE Sorting It Out," *Bridge*, December, 1992.

[SEI 92b] Software Engineering Institute, "The Role of Assessments," *Bridge*, March, 1992.

6 Process Definition

6.1 Overview

Process definition refers to the practice of formally specifying (or "modeling") a business process in a way that allows analysis of the completeness and correctness of the representation. The most often modeled process is the software development life cycle but the method is more general and can be extended to any business process. The result of process definition is a process "map" that facilitates understanding and communication, supports process improvement and management, and highlights opportunities for automation [Curtis 92]. Software development organizations often define their process through high-level, written text procedures or standards.

6.2 History/Background

Process modeling has always been a part of software development. Coding is a form of very low-level process description, and structured techniques like data flow diagramming are forms of process definition that have long been used to specify applications to be automated, and to check specifications for completeness and correctness. What is new in process definition, as it has gained momentum primarily in the late 1980s and early 1990s, is that process modeling has been turned inward on the software development process itself.

Process definition as it has evolved does not confine itself to modeling processes to be automated but aspires to specify processes enacted by humans. Movement in this direction is in preliminary stages, and there is evidence that there are still problems to be solved before process definition can fulfill its promise. Some of the problems seem serious; for example, there are fundamental disagreements about the feasibility and advisability of completely defining the development cycle at a low level of detail, because of the difficulties involved in modeling the inevitable informally structured reactions to unanticipated contingencies. Until these issues are resolved, the extent of benefits that may arise from process definition remains uncertain.

6.3 Description of Process Definition

There are really two problems being addressed by researchers in process definition:

1. The technical problem of creating a language (textual, graphical or mathematical), that will permit complete and correct specification of processes that can be rigorously, and perhaps automatically, verified.
2. The human aspects of making appropriate use of process representations, to make them helpful tools rather than bureaucratic obstacles.

Questions important to the technical line of research have to do with the ability of a specification technique to eliminate ambiguity in representation while extending modeling to ever more subtle process phenomena. Numerous techniques have been developed. Curtis et al., [Curtis 92] list five process modeling approaches:

1. Programming models attempt to use computer program-like descriptions to model general business processes.
2. Functional models are similar but emphasize functional decomposition as a means of building process depictions and lend themselves more readily to graphical analysis.
3. Plan-based models emphasize the contingent nature of many business processes and attempt to provide greater capacity for modeling contingency.
4. Petri-net models focus on the frequency and nature of the interaction of roles in a process. "Role interaction nets" aid the representation and execution of tasks that can be planned from known dependencies.
5. Quantitative models attempt to capture the dynamics of project development using mathematical expressions.

The second problem in process definition has to do more with the social aspects of the development cycle than its technical aspects. This is the human problem of how to produce process descriptions that are useful to developers, that help rather than hinder in the process of software development. To some extent, the questions important to this line of research address issues of "user-friendliness" of representations and the like – how easy is a particular process language to read, understand, and use? A weightier aspect of this problem, however, hinges on the question of how feasible and advisable it is to formalize the informal and less structured aspects of human performance in a production process, especially in response to unanticipated events. Are software production processes inherently deterministic or not? If they are not, can they be made deterministic? And, perhaps the most important but least discussed question is: should development processes be made more repeatable? Determinism and repeatability at some level leads to an enhanced capacity to improve on a stable, baseline process. Determinism imposed at a lower than appropriate level on a nondeterministic process is a definition of bureaucracy, a complex set of procedures that must be gotten around by anyone who wants to do the job right.

Process definition usually begins with a high-level process description at the level of a life-cycle model. Through use of the specification "language," process modelers explore and specify the current components of the overall process at lower and lower levels of detail. Process pieces discovered to be particularly simple, repetitive, and deterministic are candidates for automation. In representing existing processes, modelers usually also discover ways of simplifying processes. For process pieces that are neither deterministic enough to be automated nor unpredictable in the extreme, procedures can be created at a level of detail sufficient to provide guidance to those doing the development work. What to do about exceptionally nondeterministic process pieces is not yet resolved by researchers, and there are significant differences of opinion on this subject.

6.4 Experience with Process Definition

In a sense, no method is as widely deployed as process definition. Every software organization does process modeling in the course of systems development and virtually every organization has a systems life-cycle model of some sort. But process definition as the phrase is currently used (see, for example, [Feiler 92]) has a specific and rigorous definition that implies something other than what is commonly practiced. Process definition as formal, fine-grained specification of the development process is hardly practiced at all. The few attempts at such ambitious process description have been in the laboratory or in pilot programs.

Krasner et al. [Krasner 92] report the results of development and use of a process modeling tool called software process modeling system (SPMS). Although some of their findings suggest that the method is promising, they are clearly interim research results. The researchers encountered difficulties with depiction of a process model in a format useful to human developers, and with integration of the modeling system into other systems development management systems. Krasner et al. (1992) also report on experiments at the Software Engineering Institute (SEI). Three process models were defined: (1) the cleanroom software development process, (2) the IEEE P-107A configuration management process, and (3) the AT&T quality function deployment (QFD) process. The experiments identified needed enhancements to SPMS, the process definition tool. Kellner and Hansen [Kellner 89] recount use of process definition on a U.S. Air Force business process for making changes to systems currently in production (i.e., a "post deployment software support" process). Their pilot study efforts led to enhanced understanding of the process by those who performed it. The study also identified recommendations for several process improvements. These results demonstrate that process definition can be applied to beneficial effect in real situations.

6.5 Related Research

There are other fields of research that bear directly on the central problem of process definition. Organization theory has long been concerned with the degree to which formal specification of business processes is possible and advisable (see, for example, [Blau 62]). A general finding from that field is that most business processes are crucially dependent on their informal aspects, which lie outside of formally specified procedures because formal specification cannot take sufficient account of contingencies. Curtis et al. [Curtis 87] echo this point in arguing that human processes cannot be rendered adequately by inherently deterministic representations such as programming specifications. Organization theorists have also found that attempts to specify and enforce prescriptive procedures at too low a level can lead to dysfunctional behaviors that subvert the intended purpose of the specification (see, for example, [Blau 63], [Campbell 79]). It is important to notice that there are two issues here, one having to do with the feasibility of low-level specification and another having to do with whether low-level specification is a good idea even when it is possible. Organization theory suggests that such specification is possible for a relatively small subset of organizational processes and advisable for an even smaller subset.

Economists also have something to say on the issue of feasibility and advisability of low-level specification [Williamson 85]. Institutional economists argue that it is the nature of the world that consequences of acts cannot always be anticipated and that, therefore, "complete" contracts cannot be written in many situations. Complete contracts are those that specify down to a low level of detail what actions should be performed for fulfillment of the contract, including specification of responses to all contingencies. When low-level specifications can be produced easily, argue economists, then outsourcing is the recommended means of procurement. A reason why some production is coordinated within a firm is that modern production processes can almost never be specified down to an extremely low level of detail. The world is simply not that deterministic. In a nondeterministic world it is often more cost effective to delegate authority to make discretionary decisions to a loyal employee, than it is to specify the process completely.

The economics perspective raises a question that cuts to the heart of the appropriateness of process definition. Although it is fashionable in software development to assume that the benefits of process definition offset the costs, the economists' arguments can be construed as a complaint that software researchers are using faulty accounting. There is a cost of operating a business process that grows with its degree of formalization, not just due to difficulty in execution but also to the inability of the formalized process definition to adequately prescribe appropriate organizational responses. Krasner [Krasner 92] has concluded that "to be effective, future process models must also be able to explicitly represent such aspects as variability, uncertainty, intentions, assumptions, assertions, conflict, agreements, commitments, issues, pre/proscriptive actions, and nonlinear process dynamics" and that "this helps to set the research component of our agenda for the next decade." If Krasner and his fellow researchers succeed in incorporating all of these elements (i.e., variability, uncertainty, etc.) into process definitions in a way that allows them to be used at low cost, they will do far more than change the way software is developed. They will eliminate the justification for the large-scale firm and thereby dramatically change the entire business landscape. Viewed this way, the process definition agenda seems staggeringly ambitious and perhaps impossible.

This conclusion should not, however, be taken as implying that no benefit can be achieved from process definition. The method provides a valuable means of identifying opportunities for automation and for process improvement. But the ideal of completely specified processes seems unlikely to ever be attained. Some of the process specification research is, then, an example of, as Curtis [Curtis 87] puts it, looking under the lamppost because the light is better there, despite the fact that the answer to the problem is not to be found under the light.

6.6 Suggestions for Introduction and Use

Process definition as it is usually meant currently is very new. It is probably too new for a nonresearch organization to attempt a full-scale deployment of the method. However, there are some benefits that can be gained. Specifying process at even a high level can lead to better understanding of the process and to ideas for improvement. A survey of relevant literature finds that benefits are more likely to be realized from attention to the human elements of the process definition (i.e., constructing a "friendly" mode of representation) rather than to its technical rigor. Some specific suggestions can be derived from the literature.

- A graphical representation of process is much more useful to people than a textual one; this finding is fairly consistent across the literature.
- CASE tools are currently not adequate to support process modeling because they support process modeling in a way that does not reflect the complexities of nonautomated processes [Krasner 92].

- Automated tools of some sort are necessary; the volume of process description soon overcomes any manual effort at process definition [Krasner 92].
- An iterative process of interviewing and investigating to uncover the true nature of existing processes is necessary; one pass will not work [Kellner 89].
- Those interviewed during investigation of the process often, for a variety of reasons, leave things out; what is left out is often important and must be tracked down.

In practice, every software development organization must define its software development process as the baseline for continual improvement. This software development process, initially defined at a high level, must be documented, understood, and practiced throughout the organization. In most organizations today, this high-level process definition takes the form of a number of procedures often integrated within some type of software development process handbook. For organizations that are beginning this effort for the first time, it is suggested either to describe in writing how they are currently developing their products or to modify a previously existing standard (e.g., [IEEE 87]). The draft process procedures are then used to stimulate a dialogue with all software developers concerning the details of their process. The process procedures can also be used for training employees new to the organization. They may also be used to meet audit requirements for documents such as those required for ISO 9000 certification. A process description can be "descriptive," which describes the current practices, or "prescriptive," which describes an idealized process which is incrementally implemented by the organization.

6.7 How Process Definition Is Related to the Capability Maturity Model

Process definition is most commonly correlated to the level 3 key process area, organization process definition. However, even organizations at maturity level 1 should be encouraged to begin defining their process for their individual software projects as part of the implementation of the level 2 KPA, software project planning. It is necessary to have a high-level understanding of a product development process in order to do the planning for a software project. Furthermore, it is a prerequisite to have a defined process before the process can be improved.

6.8 Summary Comments on Process Definition

Process definition holds great promise to help organizations better communicate and improve their software development process. It also provides help in identifying tools that can automate the development process. Further issues remain for researchers to determine whether the various representation methods can increase understanding of the development process, as compared to more widely practiced "informal" methods of generating text-based process development standards and procedures.

6.9 References and Further Readings - Process Definition

[Blau 62] Blau, P. M., Scott, W. R., *Formal Organizations: A Comparative Approach*, Chandler Publishing Company, San Francisco, CA, 1962.

[Blau 63] Blau, P. M., *The Dynamics of Bureaucracy*, University of Chicago Press, Chicago, IL, 1963.

[Campbell 79] Campbell, D. T., "Assessing the Impact of Planned Social Change," *Evaluation and Program Planning*, Vol. 2, 67-90, 1979.

[Card 89] Card, D. N., Berg, R. A., "An Industrial Engineering Approach to Software Development," *Journal of Systems and Software*, Vol. 10, 159-168, 1989.

[Curtis 87] Curtis, B., Krasner, H., Shen, V., Iscoe, N., "On Building Software Process Models Under the Lamppost," *Proceedings of the Ninth International Conference on Software Engineering*, IEEE Computer Society, Washington D.C., 96-103, 1987.

[Curtis 88] Curtis, B., Krasner, H., Iscoe, N. "A Field Study of the Software Design Process on Large Systems," *Communications of the ACM*, Vol. 31, No. 11, 1268-1287, 1988.

[Curtis 92] Curtis, B., Kellner, M. I., Over, J., "Process Modeling," *Communications of the ACM*, Vol. 35, No. 9, 75-90, September, 1992.

[Feiler 92] Feiler, P. H., Humphrey, W. S., *Software Process Development and Enactment: Concepts and Definitions*, (CMU/SEI-92-TR-04, ADA 258465), 1992, Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University.

[Humphrey 85] Humphrey, W. S., "The IBM Large-Systems Software Development Process: Objectives and Direction," *IBM Systems Journal,* Vol. 24, No. 2, 76-78, 1985.

[IEEE 87] IEEE Software Engineering Standards, ISBN 471-63457-3, 1987.

[Kellner 89] Kellner, M. I., Hansen, G. A., "Software Process Modeling: A Case Study," *Proceedings of the Twenty-Second Annual Hawaii International Conference on Systems Science*, Vol. II - Software Track, IEEE press, 1989.

[Krasner 92] Krasner, H., Terrel, J., Linehan, A., Arnold, P., Ett, W. H., "Lessons Learned from a Software Process Modeling System," *Communications of the ACM,* Vol. 35, No. 9, 91-100, September, 1992.

[Williamson 85] Williamson, O. E., *The Economic Institutions of Capitalism: Firms, Markets, Relational Contracting*, The Free Press, NY, 1985.

7 Formal Inspection

7.1 Overview

Formal inspection is a means of removing defects from code and other structured software development work products. It is a team technique requiring strict adherence to a set of specific procedural rules. These rules distinguish formal inspection from related but less precisely defined techniques, such as peer reviews and structured walkthroughs. The terms "inspection," "peer review," "walkthrough," etc., are sometimes used interchangeably. However, "formal" inspection almost always refers to the specific procedure described here. Peer review and walkthrough often describe a larger class of defect detection processes (which may include formal inspection). "Fagan inspection" is a subclass of formal inspection in which M. Fagan's ([Fagan 76], [Fagan 86]) process is followed exactly. The purpose of formal inspection is to ensure the quality of delivered software products and improve productivity by eliminating rework.

7.2 History/Background

The formal inspection procedure was pioneered by Michael Fagan in 1972 at IBM Kingston, NY [Fagan 86]. Widespread use of inspections began in 1976 with the publication of Fagan's account of their success within IBM. In the 17 years since, many organizations have reported success in using formal inspection. The large quantity of documented experience with formal inspection provides reason to believe with high confidence that substantial benefits may be derived from the appropriate use of the method.

7.3 Description of the Formal Inspection Method

The inspection procedure consists of six discrete phases: *planning, overview, preparation, inspection meeting, rework,* and *follow-up.* The phases are carried out by a team that consists of a *moderator, author, reader,* and *recorder.* The focus of the team's efforts is a work product, for example a segment of code or design specification, from which defects will be extracted.

Responsibility for conducting the procedure falls to the moderator. In the *planning* phase, the moderator identifies the inspection team and makes logistical arrangements to support the rest of the procedure. After planning, the moderator calls the team together for an *overview.* The author of the work product to be inspected describes it to other team members and the moderator distributes copies of the work product along with inspection-defect log forms, to be used in *preparation* to record discovered defects. During *preparation,* team members review the work product for up to two hours, locating and logging defects. The reader (never the same team member as the author) prepares to present the work product to the group, making sure that he or she understands the work product.

The *inspection meeting* itself never lasts more than two hours (otherwise fatigue reduces its effectiveness). The reader presents the work product and other team members identify and discuss the defects discovered in preparation, or in the meeting itself. It is a specific stipulation of the method that discussion is limited to identifying, verifying, and recording defects; correction of defects is the responsibility of the author and discussion of correction is not permitted during the meeting. (The moderator enforces this rule.) The recorder maintains a master log of defects, which is the principal output of the meeting.

In the *rework* phase of the procedure, the author corrects defects identified on the log. After *rework*, the moderator and author jointly decide whether another iteration of the inspection procedure should occur for the work product in question. If not, the work product moves to the final phase, *follow-up*. In this phase, the moderator must verify that defects have been corrected and log a fix date for each defect.

Variations of the procedure have evolved, but most share at least the elements presented above. Common variations include the addition to the defect logging system of defect classification schemes, or slight procedural variations to account for differences in the work products being inspected (e.g., differentiation between high- and low-level design inspections, and coding inspections [Dobbins 87]).

7.4 Experience with Formal Inspection

Fagan [Fagan 86] has conducted a controlled study of the effectiveness of software inspections. In that study, design and code inspections found 82 percent of defects uncovered, leaving only 18 percent to be found in later tests. Coding productivity increased 23 percent because of the reduced need for rework. Bush [Bush 90] reports savings at NASA's Jet Propulsion Laboratory roughly estimated to be on the order of $25,000 per inspection. Fowler [Fowler 86] summarizes several Bell Laboratories studies, all of which provide support for the contention that inspections increase quality and productivity. McKissick, et al. [McKissick 84] and Peele [Peele 82] reached the same conclusion, concluding that inspection use at General Electric and First Union Corporation, respectively, resulted in cost savings and improved quality.

Summary information concerning the effectiveness of the method obtained by Fagan [Fagan 86] from a wide variety of sources includes the following:

- Inspection costs typically amount to 15 percent of project costs.

- Inspections result in the discovery of more than 50 percent of all discovered defects.
- Cases have been reported where more than 90 percent of defects were discovered by the inspection method; 80 percent defect detection is not uncommon.
- Cases of up to 85 percent savings in programmer time have been claimed by some users of code inspections.

Russell [Russell 91] reports that inspections at Bell-Northern Research (BNR) of 2.5 million lines of code resulted in defects being removed at a rate of between 0.8 and 1 defect per staff-hour. The average defect discovered in a BNR system *after* the software is released to the customer requires 4.5 *staff-days* to be removed. This suggests a return on time spent in inspections of roughly 33 to 1. This 33 to 1 ratio is not a true measure of the ratio of benefits to costs since it is unreasonable to argue that all defects not found in inspections would have reached the customer. (Some would have surely been detected in testing, for example.) Nevertheless, the magnitude of the ratio merits attention. The credibility of this 33 to 1 estimate is enhanced by Doolan [Doolan 92], who does a similar calculation for software development at Shell Research and arrives at a comparable 30 to 1 ratio. In comparing inspections with testing at BNR, Russell further claims that inspections are between 2 and 20 times more effective. He also estimates that inspections found approximately 80 percent of all errors in the BNR code.

7.5 Suggestions for Introduction and Use

Inspections succeed where related methods fail largely because of their structure [Fagan 86]. The phased procedure coupled with sharply defined roles for team members subverts nonconstructive group tendencies. For example, defensiveness on the part of the author is avoided by assigning responsibility for presenting the work product to the reader and by suppression of discussion that moves beyond what is required to identify and verify defects. The structure also maintains a low level of cognitive complexity in the defect identification task, by dividing responsibilities into manageable chunks. Modifications of the procedure that reduce its effectiveness usually fail to maintain these positive features of the inspection process.

Fagan [Fagan 86] provides the following helpful hints based on his interaction with many who have used inspections:

- Omitting or combining any of the six phases results in decreased efficiency of the method. Only the overview phase may be omitted without exposure to unacceptable levels of risk.

- The inspection meeting time limit of two hours should be strictly followed; the ability of team members to detect defects is restored after a break of at least two hours.
- Inspection team members may find a checklist of types of possible defects helpful as they prepare for and participate in the inspection meeting.
- The moderator should not be directly involved in the development of the work product to be inspected, so that he or she can maintain objectivity.
- The moderator role is extremely important; a person playing this role should act as a "player-coach"; a person with strong interpersonal skills would be ideal for this role.
- Inspection data should *never* be used in performance evaluation; such a use destroys the incentive to find defects.

Many authors ([Fowler 86], [Fagan 86], [Ackerman 82], [Collofello 87]) stress the role of training in the successful introduction of the inspection procedure into an organization. Training is important not only to ensure accurate execution of the procedure, but also to explain the advantages of the method to managers who must make available the time their employees will need to conduct inspections. In most organizations, training consists of a half-day or up to a four-day course.

Doolan [Doolan 92] and Russell [Russell 91] provide the following general suggestions concerning start-up of an inspection program.

- Choose a first project for inspection that is not too large and work actively to see that inspection does not fail for lack of motivation or commitment.
- Introduce inspection gradually, one work product at a time.
- Obtain commitment from all members of the organization, especially management.
- Constantly monitor the start-up and be prepared to make minor modifications.
- Be aware that inspection use may seem "low-tech," thus suffering an image problem compared to technology-based solution proposals.

- Be up-front about the time required for inspections, but also be prepared to show how time will be saved in the long run (during testing, for example).
- Start early on establishing "proof" of the effectiveness of the program, using inspection data.

Research at BNR by Russell [Russell 91] suggests that there is an optimal pace of the inspection meeting, at BNR, around 150 lines of code per hour. Bush [Bush 90] concludes, based on experience with inspections at NASA's Jet Propulsion Laboratory, that the number of defects found in an inspection does not significantly increase with the number of team members, thus, that three-person inspection teams can also operate effectively.

A major advantage of formal inspection is that it can be introduced at any phase of an organization's software development process. Inspections could be initially applied to reviewing test procedures or to requirements specifications depending on the current implementation phase and needs of the organization. Many organizations initially apply formal inspections to the coding phase since introduction and acceptance barriers are usually minimal. There are many commercially available training classes which can both motivate and provide the required skills for an organization to begin introducing formal inspection to their software development process.

7.6 How Formal Inspection Is Related to the Capability Maturity Model

Maturity level 3, the defined level, requires that peer reviews be a part of the development process. Since formal inspection is a type of peer review, inspection would seem to be most appropriate in organizations where the overall development process is repeatable, documented, and defined. Thus, formal inspection is a software process improvement method that can effectively meet the needs of the level 3 key process area (KPA), peer reviews, or be applied for improving the effectiveness of peer reviews. Based on the evidence of its successful application, we believe formal inspection should have value to organizations at all levels of the CMM.

7.7 Summary Comments on Formal Inspection

Formal inspection is strongly recommended to any organization wanting to improve their development process. It is one of the oldest methods described in this report. There is substantial evidence in the literature that many organizations have applied the method with good success. Formal inspection is generally recognized as an effective method for an organization to find defects earlier in the development process which results in higher quality products and reduced development and maintenance costs.

7.8 References and Further Readings - Formal Inspection

[Ackerman 82] Ackerman, A. Frank, Amy S. Ackerman, and Robert G. Ebenau, "A Software Inspections Training Program," *COMPSAC '82: 1982 Computer Software and Applications Conference*, Chicago, IL, Nov. 8-12, pp. 443-444, IEEE Computer Society Press.

[Bush 90] Bush, Marilyn. "Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory," *12th International Conference on Software Engineering*, 1990, pp. 196-199, IEEE Computer Society Press.

[Collofello 87] Collofello, James S. "Teaching Technical Reviews in a One-Semester Software Engineering Course," *ACM SIGCSE Bulletin,* Vol. 19, No. 1, Feb. 1987, pp. 222-227.

[Dobbins 87] Dobbins, J. H. "Inspections as an Up-Front Quality Technique," *Handbook of Software Quality Assurance*, G. G. Schulmeyer and J. I. McManus, eds., pp. 137-177, NY: Van Nostrand Reinhold, 1987.

[Doolan 92] Doolan, E. P. "Experience with Fagan's Inspection Method," *Software – Practice and Experience*, Vol. 22, No. 2, Feb. 1992, pp. 173-182.

[Fagan 86] Fagan, Michael E., "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, Vol. 12, No. 7, July, 1986, pp. 744-751.

[Fagan 76] Fagan, Michael E. "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, Vol. 15, No. 3, 1976, pp. 182-211.

[Fowler 86] Fowler, Priscilla J., "In-Process Inspections of Workproducts at AT&T," *AT&T Technical Journal*, March/April 1986, pp. 102-112.

[McKissick 84] McKissick, John Jr., Mark J. Somers, and Wilhelmina Marsh. "Software Design Inspection for Preliminary Design" *COMPSAC '84: 1984 Computer Software and Applications Conference*, Las Vegas, NV, Jul. 1984, pp. 273-281.

[Peele 82] Peele, R. "Code Inspections at First Union Corporation," *COMPSAC '82: 1982 Computer Software and Applications Conference*, Chicago, IL, Nov. 8-12, 1982, pp. 445-446, IEEE Computer Society Press.

[Russell 91] Russell, Glen W. "Experience with Inspection in Ultralarge-Scale Developments, *IEEE Software*, Vol. 8, No. 1, Jan. 1991, pp. 25-31.

8 Software Measurement and Metrics

8.1 Overview

"Software measurement" describes the use of quantitative indicators to learn about and improve the software development process. A software metric is strictly defined as a unit of measure used in software measurement ([Zuse 90], [Fenton 91]), but the phrase "software metrics" is often used more broadly to describe the process of measurement as well. The purposes of software measurement are to provide:

1. Status information on individual projects to help planners anticipate problems.
2. Information about the development process that can be used to make improvements in the process.
3. A base of statistical information that can be used to predict project duration, cost, etc.
4. Feedback to developers to help them understand how to change their own performance to produce better results.

8.2 History/Background

Software measurement arose primarily from four technology trends that began in the 1970's [Möller 93].

1. There was significant research into ways of measuring code complexity based upon graph-theoretic concepts. Complexity was assumed to be associated with development difficulty and was easily quantified by automated means.
2. Attempts arose to quantify factors associated with project cost so that the cost of future projects could be estimated based on statistical comparisons to similar projects.
3. Notions of quality assurance began to filter into the software world, resulting in the creation of methods to detect, track, and prevent faults during various stages of the software development cycle.
4. The software development process itself began to take on a more structured form that emphasized control of development resources across the various stages of development. Such control required the development of quantitative measures of resources.

Because software measurement describes such a broad set of practices, it is difficult to make blanket comments on the success of its use. However, there seems to be many claims of successful implementation of measurement across a variety of organizations.

8.3 Description of Software Measurement

Software measurement is associated with a wide variety of techniques that share a dependence on the gathering of quantitative data. A primary reason for this wide variation of techniques is that measurement application is merely a tool for identifying actions for improving the software development process. Measures defined for a specific organization must be derived from the goals of that organization, and be tailored to function within the software development process and culture unique to that organization. Since space provided here does not allow exhaustive accounting of all measurement-related processes, we will merely enumerate and briefly describe common categories of measurement. Those wishing a more detailed treatment are invited to consult Möller and Paulish [Möller 93] or any of a number of other references on the subject (e.g., [Grady 87], [Jones 91]).

8.3.1 Example Measures

Size is a commonly applied measurement category. Size can be measured in units such as function point counts, lines of code, or effort. It is a useful measurement category for helping to establish system development effort, schedules, and cost. It is also often used as a "normalizing" factor for measures within other categories described below (e.g., faults found per number of thousand lines of code).

Product quality characterizes the quality level of the software product or service itself. Some measures can be "interim" for providing insights into the final product quality or complexity. Examples of measures in the product quality category could include the number of faults detected and counted during various phases (e.g., system testing) of the development process. Earlier sections of this report provided descriptions of specific methods of detecting and counting software defects. Measurements of defects that allow determination of where they originated (i.e., in what phase of development), where they were detected, etc., are of obvious use in guiding the introduction of quality improvement actions and tracking their effects. Those who count defects must solve problems of standardizing means of counting, and must also be careful to avoid misinterpreting changes in defect levels; there is, for example, danger in interpreting a defect rate decrease as the result of process improvements if the complexity of applications being developed was reduced at the same time. To be confident of the interpretation of defect rates (as well as many other indicators), one must strive to capture all independent variables that may have an effect on that rate. Nevertheless, defect detection has been used successfully in many organizations.

Process quality characterizes the maturity of the process within which the software product is developed. These measures can be obtained during the product development for providing real-time feedback to software project management. An example of this type of measure would be the management of development team productivity. For productivity, one usually compares the rate of output to the effort expended in production of that output. The two most common output measures are lines of code (LOC) and function points (FP) [Albrecht 79]. Effort is most commonly measured in staff-hours. Like all measures of productivity, software productivity metrics are beset by a variety of complications having to do with the reliability of the measurement mechanism, lack of standardization (e.g., division 1 counts function points in a different way than division 2), and strategic behavior by developers ("If it's lines of code you want, it's lines of code you'll get...").

Environment characterizes the organizational resource environment within which the product is developed. Experience within this category is least mature to the point that the measures are often referred to as "soft factors." Measures in this category often try to capture characteristics of development teams or software producing organizations such as staff size, staff turnover, morale, skills distribution, etc.

It must also be pointed out that measurement application is an "accompanying technology." This means that measurement must be used with other methods for improving the software development process. The collection of measurement data has no intrinsic value if the trend of the data is not analyzed to provide insights about what to do to improve the development process. Thus, it is not measurement that has an impact on the development process, but rather the actions to improve the development process that were selected as a result of analyzing the trend data. Thus, it is most common to hear about measurement being used along with other described process improvement methods (e.g., inspection, DPP, estimation).

8.4 Experience with Software Measurement

There is a large body of documented experience with software measurement which suggests that its use is potentially beneficial. Möller and Paulish [Möller 93] provide a series of case studies of successful applications. Grady and Caswell [Grady 87] describe the metrics program at Hewlett-Packard, which is widely acknowledged to be among the best. However, it is difficult to cite meaningful quantitative evidence of the benefits due to measurement programs because "software measurement" describes such a broad set of practices and because its impact is inherently difficult to evaluate empirically.

While there is a clear consensus among most in the software industry that metrics are generally beneficial, there are vocal exceptions (e.g., [Evangilist 88]). There have been reports of unsuccessful measurement programs, but these are not often found in the literature; this fact creates a sampling bias in our database regarding the technique (i.e., we mostly hear about successes). Without an unbiased sample, it is difficult to systematically attribute failures of measurement programs to particular causes. The conclusion one would hope to reach is that software measurement is potentially beneficial for most organizations, that benefits follow when certain specific practices or procedures are followed, and that failures were due to errors in implementation.

8.5 Suggestions for Introduction and Use

Möller and Paulish [Möller 93] suggest a seven-step approach for the introduction of a measurement program:

1. Establish and document a baseline software development process, which will be improved dynamically over time.
2. Identify goals that are derived from and support strategic business objectives.
3. Assign responsibility for introduction of the program to an individual who has the ability, commitment, and stature within the organization to act as a persuasive advocate.
4. Do initial research to determine what might be measured in the organization, by surveying members of the organization or using a formal assessment process (e.g., as described in [Paulk 93]).
5. Define a basic set of metrics for measuring progress toward goals.
6. Introduce and communicate the metric program to the organization such that high visibility and cooperation are achieved.
7. Identify the metrics reporting and feedback mechanisms such that actions for improving the software development process can be determined and implemented.

Möller and Paulish [Möller 93] also provide the following hints concerning introduction and use of a measurement program:

- Overcome lack of acceptance by educating developers about the purpose and substance of the measurements; assure them that any additional work required will be supported by their managers (and see to it that this is demonstrably true); make the benefits of the program clear to all, but diffuse expectations of "quick-fixes."
- Explicitly decouple the measurement program from individual performance appraisal.
- Work to maintain momentum beyond the introduction of measurement; loss of momentum will threaten when the excitement subsides and the hard work begins.
- Take steps to see that any tools required by the program are made available.
- Make sure that goals are well defined and that process improvement suggestions are acted upon quickly.
- Emphasize teamwork and joint responsibility for solving problems, rather than attributing blame.

8.6 How Software Measurement Is Related to the Capability Maturity Model

Software measurement is used at stages of the maturity model above level 1. Higher stages require more sophisticated measurement systems. At the fourth and fifth levels, respectively, "process and products are quantitatively understood and controlled using detailed measures," and "continuous improvement is enabled by quantitative feedback" [Paulk 93]. For more details on the particular metrics that can be used at each level of the CMM, see [Baumert 92].

Although measurement is most emphasized within the CMM level 4 (managed) as part of the KPA quantitative process management, it is a tool that can be effectively used at all levels of the CMM in varying degrees. This can be illustrated in Figure 8-1 which identifies a higher level of measurement application sophistication for organizations at higher levels of the CMM. It is suggested that level 1 organizations not consider using measurement until they have defined and documented their high-level software development process. Once a process is documented and is being followed, an organization can begin collecting primary data that will be useful to build project management skills for the level 2 KPAs software project planning and software project tracking and oversight. Primary data could include measures for such basic categories as size, effort, schedule, and defects. At level 2, the organization uses the primary data for estimating and tracking projects. At level 3, the organization uses product quality data to support their process. Organizations at level 4 implement process measurement and analysis. At level 5 measurement is used to support DPP and provide feedback for process change management. Organizations at level 5 can also perform action alternative tradeoffs based upon benefit measurements.

Investment in measurement must be made to increase an organization to a higher level. This investment requires building skills, and integrating more sophisticated use of measurement for data collection and analysis coupled to the organization's goals and process. In general, the investment necessary to initiate measurement activities and barriers to overcome are most difficult for an organization moving from level 1 to level 2 and from level 3 to level 4 [Humphrey 89].

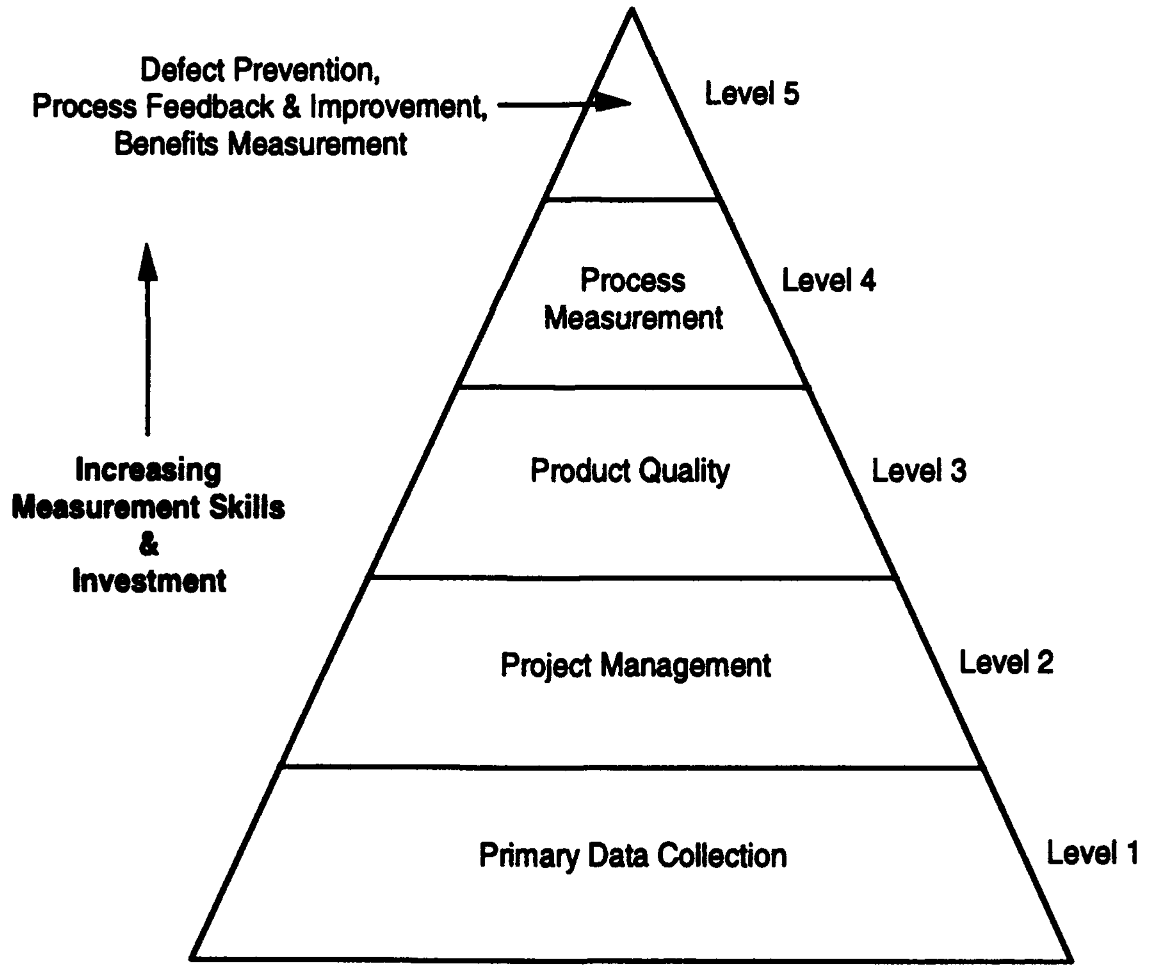

Figure 8-1. Measurement Activities for Organizations at Various CMM Levels

8.7 Summary Comments on Measurement

Measurement is a fundamental skill for organizations wishing to improve their software development process. Measurement can be applied at all levels of the CMM. Practitioners of measurement must remember that collection of data by itself has minimal value. What is important is how the data are used to continually improve the software development process. Measurement is thus widely used along with other methods of software process improvement.

8.8 References and Further Readings - Measurement

[Albrecht 79] Albrecht, A.J. "Measuring Application Development Productivity," *Proc. of the Joint SHARE/GUIDE Symposium*, pp. 83-92.

[Austin 92] Austin, R., Larkey, P. "The Unintended Consequences of Micromanagement: The Case of Procuring Mission Critical Computer Resources," *Policy Sciences*, 1992.

[Baumert 92] Baumert, John H. *Software Measures and the Capability Maturity Model* (CMU/SEI-92-TR-25) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992

[Boehm 81] Boehm, Barry W. *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Evangelist 88] Evangelist, Michael, "Complete Solution to the Measurement Problem," *IEEE Software*, Jan. 1988.

[Fenton 91] Fenton, Norman E. *Software Metrics: A Rigorous Approach*, Chapman & Hall, London, 1991.

[Grady 87] Grady, Robert B., Caswell, Deborah L., *Software Metrics: Establishing a Company-Wide Program*, Englewood Cliffs, NJ: Prentice-Hall, 1987.

[Humphrey 89] Humphrey, W. S. *Managing the Software Process*, Addison-Wesley, Reading, MA, 1989.

[Jones 91] Jones, Capers, *Applied Software Measurement*, New York: McGraw-Hill, 1991.

[Möller 93]	Möller, K.-H., Paulish, D.J. *Software Metrics: A Practitioner's Guide to Improved Product Development*, IEEE Computer Society Press, Los Alamitos, CA, 1993.
[Paulk 93]	Paulk, Mark C. et al, *Capability Maturity Model for Software, Version 1.1*, (CMU/SEI-93-TR-24, ESC-TR-93-177), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, February 1993.
[Zuse 90]	Zuse, H. *Software Complexity: Measures and Methods*, de Gruyter, 1990.

9 Computer Aided Software Engineering (CASE)

9.1 Overview

Computer aided software engineering, usually denoted by its acronym CASE, refers to automation of the software development process. The central premise of CASE is that the fundamental structure of a software system is present in abstract models of its data organization and process flows, which typically exist very early in the development process. If an abstract model can be created in a standardized (perhaps diagrammatic) language, then lower level elements of design, such as physical database definitions and code, can be automatically generated from the model. Benefits from CASE derive from the fact that once an automated system generation capability is in place, changes to the system are easier. Changes are made to the abstract model, then a replacement system is automatically generated. Development becomes a process of creating the model, generating and testing a system, adjusting the model, regenerating and retesting, and so on. Taken to its logical conclusion, CASE could make coding as we now know it today virtually obsolete, just as third generation languages dramatically reduced the prevalence of machine coding. The resulting productivity gains are obvious, but equally dramatic quality improvements could result as well, because of decreased opportunities to inject defects and automated tests to detect defects that do exist.

9.2 History/Background

What is now called CASE began in the mid-70s when Dan Tiechroew and his colleagues at the University of Michigan developed a user-oriented requirements specification tool called PSL/PSA, which had a limited capability for detecting specification errors but did not permit the generation of lower level specifications [Barros 92]. Since then, a variety of products designed to automate segments of the development cycle have evolved for many different computing environments.

Results of using CASE are mixed. Some projects experience significant benefits. In others, benefits are not realized and CASE technology becomes "shelf-ware." Some of the reasons for the differences in experiences are summarized below. For now, CASE is a means of improvement to be used judiciously, by relatively mature software organizations. However, the greatest payback from CASE is probably yet to come, as technology and software organizations evolve and improve.

9.3 Description of CASE

9.3.1 Examples of CASE Tools

Automation of any aspect of the development cycle can be called "CASE." Business planning, data and process modeling, database design, and code generation tools might all be considered CASE tools. The makers of some simple drawing tools have even claimed the title. As of 1990, there were more than 160 vendors claiming to sell CASE tools [Nejmeh 90]. Most commonly, CASE is used to refer to tools that support software design and analysis methods.

CASE tools that are designed to support specific phases of the software development process are called vertical CASE tools. Some example vertical CASE tools are given in Table 2, which is reproduced from an internal Siemens report.

Table 2: Vertical CASE Tools

Phase	UNIX	DOS	Macintosh
Concept	FrameMaker, Interleaf	Word	Word, Claris
Requirements	StP, Teamwork, StateMate, Power Tools	EasyCASE, Power Tools, System Developer I	HyperCard, MacAnalyst, OOA, Power Tools, TurboCase
Design	StP, Teamwork, StateMate, Power Tools	EasyCASE, Power Tools, System Developer I	MacDesigner, OOD, Power Tools, TurboCase
Construction	cc, CC (C++), CodeCenter (C), ObjectCenter (C++), Power Tools, Teamwork	Borland C++, Microsoft C, Power Tools, Turbo C, Turbo C++	Mac Programmer's Workshop, Power Tools, Think C
Integration & Test	Power Tools, SmartSystem, Teamwork	Power Tools	Mac Programmer's Workshop, Power Tools
Maintenance	StP, Teamwork, Hindsight, BattleMap, SmartSystem	Power Tools	Mac Programmer's Workshop, Power Tools

CASE tools that are used to support an activity that occurs during multiple or all phases of the software development process are called horizontal CASE tools. Some example horizontal CASE tools are given in Table 3, which is reproduced from a Siemens internal report.

Table 3: Horizontal CASE Tools

Activity	UNIX	DOS	Macintosh
Configuration Mgt.	rcs, sccs, TeamNet	PVCS, Version Manager	Mac Programmer's Workshop
Documentation	FrameMaker, Interleaf	Word	Word, Claris
Project Mgt.	GECOMO Plus, Pertmaster Advance	Microsoft Project, Time Line	Project Scheduler 4, MacProject, Microsoft Project
Measurement & Quality Assurance	GECOMO Plus, Battlemap, SIZE Plus, UX-Metric	Checkpoint, PC-Metric, Software Metrics Repository	MC-Metric

9.3.2 Upper and Lower CASE

The application of CASE to requirements definition, design, and analysis—sometimes called simply "upper CASE"—is initiated by the request for a specific system to be built. Requirements analysis frequently proceeds in a forum much like business planning, with a facilitated cross-functional group process used to identify, group, and prioritize requirements, and a CASE tool used to document the process. When requirements have been defined, analysis often begins with analytical data and process models. In the futuristic ideal case, a data model for the entities involved in the system already exists in the organization's library of data models, and can be incorporated and tailored for the system at hand. Few organizations are currently so advanced; thus, the data model does not exist and must be created in a diagrammatic specification language. Similarly, in the ideal situation, a generic process for the system will already exist, but usually this must also be created, again in a specification language. The models are then linked, and cross checked automatically (to ensure, for example, that processes do not require data elements that are not present).

Eventually, CASE technology is expected to evolve to a point where most of the work will be done during analysis. Low-level design will involve various decisions about the physical implementation of the system (such as how a database will be tailored to fit the physical database environment). Thus "lower CASE" tools are required to refine data and process models into physical implementations from which code can be generated. Designers, working at design workstations, carry out this work and operate the generation facilities which produce code. In addition, there exist upper and lower CASE tools, which are capable of producing complete applications for specialized domains (e.g., system development language (SDL) for telecommunications, information engineering for management information systems (MIS)).

9.4 Experience with CASE

There are no long-term studies on the economics of CASE, because the industry is barely 10 years old. In fact, over half of all CASE users have less than three years experience with it [Jones 92]. Users of CASE seem to experience productivity gains, but not always dramatic ones. Sullivan [Sullivan 90] claims that there are only two studies that show a relationship between CASE and project productivity. One of these, by Lempp and Lauber, 1988, reports a 9 percent cost savings immediately, but a 69 percent savings on maintenance costs. Since maintenance costs are a considerable portion of system costs over the life of a system, Sullivan concludes that the 9 percent savings reported by Lempp and Lauber is probably an underestimate. Savings on maintenance are a consistent finding in the literature on CASE [Jones 92].

Necco, et al. (1987) found that systems designed using CASE were ranked higher than non-CASE systems in a number of important development performance categories (e.g., timeliness of delivery). Binder [Binder 90] reports that according to his survey, software developers that use CASE claim an average 28 percent perceived (i.e., subjectively estimated, not measured) increase in overall productivity. However, as Jones [Jones 92] points out, no controlled experimental studies have been done. He finds, based on his own survey data, that in the first year of CASE use, productivity actually decreases (probably due to learning effects). Jones estimates a payback period of between two years under very favorable conditions (i.e., when introduction of CASE and requisite training coincide with the start of several important projects), and never as when a CASE system is abandoned.

While there are no hard data to support their claim, users of CASE believe that quality improvements are one of the main advantages of CASE [Sullivan 90], [Jones 92]. The reason for this persistent perception seems to be that CASE-produced work products are easier to review and test [Jones 92].

While benefits do seem to accrue to those who call themselves users of CASE, it is clear that there is a sampling bias. Organizations that begin to use CASE and do not succeed, probably do not call themselves CASE users; their experiences are not reported in studies of CASE users. This fact is not lost on those who have studied CASE, and consequently, there are some studies that attempt to identify the differences between CASE users and nonusers.

It appears that successful CASE users are well trained [Binder 90], [Jones 92]. Without effective training, failure rates exceed 50 percent [Jones 92]. Organizations where CASE has been successful are already using sound and sophisticated methods (e.g., entity relationship modeling, Yourdon process modeling) methods, and tend to use CASE in conjunction with intensive customer-involvement methods such as joint application design (JAD) [Sullivan 90]. Also, not surprisingly, successful CASE users had been using CASE for a relatively long time [Binder 90]. Culver-Lozo and Glezman [Culver-Lozo 92] recount use of CASE tools on 15 projects, 6 of which abandoned CASE midway through; the differences they observe between successful and unsuccessful CASE users echo those above.

9.5 Suggestions for Introduction and Use

Many of the above reported experiences suggest advice on adopting CASE:

- Introduce and train everyone in the design and analysis methods first.
- Make sure all analysts who will use CASE are well trained on the tool.
- Use CASE in conjunction with methods that promote intensive interaction between customer and analyst.
- Be prepared for a learning curve; do not expect immediate results.
- Make sure managers understand that CASE will change development, require learning time, and perhaps cause productivity losses in the first year.

CASE users provide the following additional tips [Nash 92]:

- Using CASE on just one project is a waste of money because start up costs will wipe out any savings; to obtain real payback, CASE must be deployed more broadly.
- Start with projects that have somewhat flexible deadlines because of the learning curve required.
- Prepare staff to relearn their jobs; developing with CASE is very different than developing in old-style mainframe environments.
- Choose analysts with business sense, who can translate the ambiguous into the structured language of the CASE tool.
- Make sure the vendors or experienced third-party consultants are available for hand-holding.

- Use CASE in conjunction with project management tools and techniques, because CASE projects grow quickly and in ways to which analysts are unaccustomed.
- Establish a measurement system that you can use to determine the benefits of CASE.

Similarly, an internal Siemens report identifies the following CASE adoption issues:

- Gain control of the software development process first, then apply CASE.
- Consider new methods along with the tools, automating a mess will only result in an automated mess.
- Before selecting a CASE tool, select a development methodology for the project.
- Be aware of and plan for the need to manage organizational change.
- Train management and staff on both the new methods and CASE tools.
- Plan for all costs such as those to purchase the tools and training, as well as personnel costs associated with learning and tools support.
- Manage expectations; productivity may decrease during the learning curve, and gains may be minimal.
- Use measurement to monitor progress and impact of CASE.

9.6 How CASE Is Related to the Capability Maturity Model

As has been observed already, CASE is most likely to succeed in software organizations with mature, well-defined processes. At the least, a software development process must be defined, which would place CASE application at level 3. CASE use would be primarily within the level 3 KPAs of organization process focus and software product engineering. Horizontal CASE tools can be used to support the implementation of level 2 KPAs such as software configuration management, software quality assurance, and software project tracking and oversight. Within application domains where fully automated solutions exist, it is recommended that organizations at any process maturity consider applying such tools (e.g., SDL for telecommunications, information engineering for MIS).

9.7 Summary Comments on CASE

CASE seems to hold some promise for changing the way software is created. The technology has advanced to the point where successes with CASE are beginning to be significant. This will probably improve in the coming years. However, it is clear that CASE is not an off-the-shelf panacea, a miraculous technology which needs only to be purchased to provide return on investment. Organizations that successfully use CASE are, almost without exception, already successful in the use of structured methods of requirements definition, analysis, and design. Before using CASE, an organization must get its process house in order. Then, it must weigh the suitability of its applications environment for this not entirely mature improvement method. Sophisticated users with appropriate applications will probably experience success with CASE, certainly in the long run, if not right away.

9.8 References and Further Readings - CASE

[Barros 92] Barros, Oscar. "A Pragmatic Approach to Computer-Assisted System Building," *Journal of Systems and Software*, Vol. 18, 1992.

[Binder 90] Binder, Robert. "A Model of CASE Users," *CASE Outlook*, No. 1, 1990.

[Culver-Lozo 92] Culver-Lozo, Kathleen, Glezman, Vicki. "Experiences Applying Methods Supported by CASE Tools," *AT&T Technical Journal*, November/December 1992.

[Jones 92] Jones, Capers. "CASE's Missing Elements," *IEEE Spectrum*, June, 1992.

[Nash 92] Nash, Kim S. "Words from the CASE-wise: Think big, know your vendor," *ComputerWorld*, Vol. XXVI, No. 43, October 26, 1992.

[Nejmeh 90] Nejmeh, Brian A. "Designs on CASE," *Unix Review*, Vol. 6, No. 11, 1990.

[Sullivan 90] Sullivan, Patrick J. "A Study of CASE Early Adopters," *CASE Outlook*, No. 1, 1990.

10 Interdisciplinary Group Methods (IGMs)

10.1 Overview

Interdisciplinary group methods (or IGMs) is our collective term for the various forms of planned interaction engaged in by people of diverse expertise and functional responsibilities working jointly toward the completion of a software system. Many IGMs are intended to structure group interaction through use of rules, technologies, or modes of organization, to improve group performance. Examples of such methods include joint application design (or JAD), certain forms of rapid prototyping, concurrent engineering, and quality circles. The purpose of IGMs is to maximize the advantages of group work, which result from bringing numerous and diverse perspectives to bear on a problem, while minimizing group work's undesirable side affects, such as real or imagined pressures to conform.

10.2 History/Background

IGMs arose from suggestions that gained popularity in the late 1960s that groups might perform some tasks better than individuals. Application of this idea took numerous forms, but usually involved structuring interaction between individuals in face-to-face meetings, or providing technological support for their interactions. The ideas were extended to software design in the form of JAD as promoted by IBM, and a variety of emerging technologies for electronic mail and conferencing. Despite the relatively long history, evidence of the success of these methods remains mixed. Part of the problem is that variation in methods and the group task undertaken makes it difficult to compare "apples to apples." Success in use seems conditional on a number of factors, not all of which are known. Consequently, all recommendations of these methods must be qualified with the phrase "in appropriate circumstances." In appropriate circumstances, however, there is evidence that these methods can be highly successful.

10.3 Description of Interdisciplinary Group Methods

Three loose and overlapping categories of methods can be defined: group dynamics facilitation, groupware methods, and team organization.

- *Group dynamics facilitation* includes methods such as nominal group technique (NGT) [Delbecq 75], [Hegedus 86], and related software-specific techniques that have arisen to define system requirements (like JAD, [Corbin 91]) or to estimate project cost and schedule [Taff 91].

- *Groupware methods* are those which make use of software tools that support group interaction; these tools range from group conferencing and electronic mail software to certain varieties of rapid prototyping software.

- *Team organization* describes the practice of staffing project groups with personnel of mixed expertise and functional responsibilities to improve communication and foster cross-pollination of ideas. Concurrent engineering is a commonly used phrase that refers to such a type of team organization.

Each of these methods is explained further in the following sections. Many specific techniques overlap two or more categories.

10.3.1 Group Dynamics Facilitation

This is perhaps the oldest category of IGMs and the most generic. Davis [Davis 69] argued that there were advantages to resolving problems within groups, not least of which were the different insights and knowledge bases that could be brought to bear on the problem. By extension, an interdisciplinary group would offer special advantages because of its corresponding wider diversity of insights and knowledge bases. But in the early 70s, behavioral researchers believed that many of the benefits of group problem solving were not realized in real groups because of undesirable characteristics of group interaction. Most significant among these were group pressures to avoid expressing deviant ideas or offering suggestions that would be deemed not in keeping with the theme of the discussion. Groupthink, as studied by Janis [Janis 72], was recognized as a particularly onerous problem in which members of a group who did not agree with the majority were considered disloyal and ostracized by the group, especially in crisis situations. The desire to realize the benefits without the costs of group interaction lead to the development of the nominal group technique (NGT) from which many group software design methods are descended.

NGT involves use of a facilitator who enforces certain rules of interaction in the group. The rules are intended to eliminate pressures to conform and result in the five unique features of NGT, which are recognizable in methods like JAD today. The five features are: (1) individual work that precedes group discussion, (2) round-robin reporting to communicate ideas among the group members, (3) a period of largely unstructured group discussion, (4) a polling procedure used to converge on a specific solution, and, (5) some face-to-face individual and group work [Hegedus 86].

Numerous techniques similar to NGT have been developed to help software developers improve requirements gathering and analysis. These techniques include JAD, WISDM, Rapid Analysis, CONSENSUS, Accelerated Analysis, The Method, and many similar methods by other names. All use a facilitator who leads the group through steps of idea generation, clarification, discussion, and decision. In some, subgroups work together during breakout periods and return to the larger group to report their results. Sessions last anywhere from one to two-and-one-half weeks, and involve a dozen or so participants, mostly from the user community. Development personnel are available to answer technical questions but play a secondary role. The idea is to have the users design their own system specification. Some of the methods produce complete requirements and analysis specifications, sometimes including data flow diagrams and data organization tables. The duration of the requirements gathering and analysis phases of the development cycle are often dramatically shortened as a consequence.

10.3.2 Groupware Methods

Groupware is an exceptionally broad category of software tools. It ranges from technologies that facilitate group communication (like electronic mail and conferencing), to sophisticated computerized meeting facilities, to advanced rapid prototyping technologies, or any combination of these. The common feature of groupware methods is that they all constitute conscious attempts to take advantages of the new capabilities provided by the software tools. Potential advantages are due not only to modes of interaction made possible that formerly were not, but also to the ability to impose structure on interaction through use of the technology. Where group dynamics facilitation used a facilitator to enforce corrections to group process, groupware methods can substitute technological constraints. Technology can be designed, for example, to equalize participation among group members or to allow anonymous voting on an issue.

One sort of groupware is often used in support of group dynamics facilitation. During a group session much information is generated and it is convenient to capture the information by using an automated tool. If the tool can interface with a CASE tool, then the development process is made even more streamlined. Having the output of such a session in a format where it can easily be modified, version controlled, and reproduced is a great advantage in many situations.

Computerized meeting facilities take groupware one step further. These "group decision support systems" or GDSSs often allow each person access to an independent computer [Dennis 88]. Group members exchange control of a commonly visible screen usually mounted on a wall, and may be able to communicate to each other privately via the keyboard. Obviously there are a great number of possible configurations of such rooms, many of which might hold promise for a software development facility. Elaborations of this concept include meeting rooms where not all members of the group are in the same location, but still are able to communicate and see a common work surface.

Particularly advanced forms of groupware methods might actually involve users in construction of a prototype system. "Fourth generation" prototyping systems which include facilities for rapid construction of user interfaces and automatic coding from specifications are not far from this goal [Gronbaek 88]. One can easily imagine a combination of a JAD-like method with prototyping tools in an automated meeting facility, which would allow users to interact with prototypes and specify further development of the system at each stage of development. Such a facility is currently under development at the University of Michigan's Cognitive Science and Machine Intelligence Laboratory [Katterman 90].

10.3.3 Team Organization

A team is a group of people with particular expertise(s) assembled to perform a specific task. Team organization describes ways of choosing the composition and organizational structure of teams to improve the likelihood of a favorable outcome. One useful and common team organization is interdisciplinary and linked together by dense informal relationships. Concurrent engineering, sometimes referred to as simultaneous engineering ([Dumaine 89], [Gordon 89], [Schmelzer 89]), calls for constructing a team composed of all the different functions required in the development of a product and involving all of them simultaneously in a project from beginning to end. Many benefits have been realized from such a team organization: problems that arise are more often anticipated and coordination is improved throughout the production cycle. It seems likely that especially large or complex projects, where coordination and unexpected problems are common, would benefit most from attention to team organization. In particular, recent attention is being paid to software projects in which software is embedded within hardware which is also being designed for the first time. Techniques applied to such systems are collectively referred to as hardware/software codesign.

Other sorts of team organization have played important roles in software process improvement. Quality circles (QCs) have been implemented in many organizations. QCs are a combination of team organization and group dynamics facilitation in that they usually follow a structured problem solving routine that involves data analysis, cause-and-effect diagramming, and elementary statistical plotting.

10.4 Experience with Interdisciplinary Group Methods

IGMs are not a single specific method. There are many variations and, because IGMs are specifically designed to affect the delicate balance of human social interaction, the differences matter. It is, therefore, exceedingly difficult to make general statements about the usefulness of IGMs. In some contexts they are spectacularly successful, and in others they are dismal failures. Successes are necessarily over-represented in the trade literature (people usually do not report failures), but the mixed results from the research literature provides reason to believe that the failures are out there.

10.4.1 Group Dynamics Facilitation

There has been considerable experimental research done on NGT for a variety of tasks. Conclusions about the performance of groups using the method relative to the performance of individuals and unstructured groups are mixed, but some general principles do seem to emerge [Hegedus 86]. One of these focuses on the nature of the task being undertaken. In particular, whether the task is easily divisible into manageable chunks seems to make a difference. If not, "the individual-group decision sequence in the NGT may compound, rather than reduce, the complexity of such tasks" ([Hegedus 86], pg. 547). Use of NGT-like methods (including JAD and its siblings) on more unitary evaluation and decision making tasks is likely to produce poor results. Another finding is that NGT participants often express satisfaction with the method whether or not it produces better outcomes [Gladstein 84]. Thus NGT and similar methods have the potential to become popular without doing much good.

Fortunately, many software requirements definition and analysis tasks are divisible into subtasks. There is evidence that when group sessions are restricted to the purposes for which they were conceived (i.e., requirements definition and analysis for a particular system), the chances for success are good (see [Corbin 91], for example). Organizations considering using a similar technique for less standard tasks should proceed carefully. Consultants who specialize in facilitation of group sessions can sometimes advocate the "I've got a hammer and all of your problems are nails" syndrome. Other uses of the method are possible, but common sense and the above stated principle of suitability of task should be employed in extending the method to other types of tasks.

10.4.2 Groupware Methods

In the area of groupware methods, "apple-to-apple" comparisons are exceptionally hard to find. So many of the technologies that permit these methods are just emerging that it is impossible to generalize. Electronic mail and conferencing facilities are widely used and have been studied [Sproull 86], and computerized meeting facilities have been the subject of some research (e.g., [Nunamaker 88], [Dennis 88], [DeSanctis 87]), as has rapid prototyping (e.g., [Gronbaek 88]). But very few general principles have emerged. One of the case study sites in the Siemens/SEI Measuring Software Process Improvement Methods Project is utilizing groupware methods for multinational product development among teams in Germany and the USA. The following paragraphs describe experience with the most common groupware methods – electronic mail, computerized meetings rooms, and rapid prototyping.

Electronic Mail

Research on electronic mail has revealed that people do change their patterns of social interaction in response to new technologies. For example, electronic mail communication is without many of the cues of social context that are present in verbal communication, like tone of voice, and this has effects on the nature of interaction via the electronic medium. Among the interesting findings is that people are more likely to "flame" – i.e., to use strongly worded statements, including profanity – when using electronic mail than they would in face-to-face communication [Sproull 86].

Computerized Meeting Rooms

Computerized meeting room research has shown that participation can be made more equal by use of technology [Zigurs 88]. Results concerning whether task outcome is improved are mixed. As with NGT, satisfaction with the method seems high, but satisfaction is sometimes not correlated with improved performance. Like NGTs, groupware technologies may be popular without doing much good.

Rapid Prototyping

Few results are available on rapid prototyping. Gronbaek [Gronbaek 88] conducted a limited empirical study in which he found that horizontal prototypes, which had the surface appearance of a system but little functionality, were inferior to vertical prototypes, which provided a narrower range of interfaces but more functionality, at engaging the user's interest.

10.4.3 Team Organization

Team organization, especially concurrent engineering, is widely credited with significant improvements in product development over the past decade and a half. Team organization is usually customized to the task, which compromises the relevance of experimental research on small group work. For this reason, most of the evidence of the success of team organization is anecdotal, based on a few famous success stories.

One of the most famous examples of successful team organization is "Team Taurus," which designed the Ford Motor Company's highly successful sedan. Ford combined an interdisciplinary group of designers, process engineers, tool builders, marketers, sales people, promotion staff, and research and development (R&D) people in a team that worked on a design together from drawing board to dealer showroom. Many activities that had traditionally been performed in sequence (design then manufacturing then marketing) were done in parallel with close coordination among team members engaged in different activities (hence the phrase "concurrent engineering") [Boudette

90]. There is evidence that Japanese auto firms have worked in similar teams for many years [Ouchi 81], [Liker 93]. Teams are becoming more prevalent in western industry overall. Siemens Automotive Division of Newport News, Va. has reorganized completely into teams. There are no longer functional departments (e.g., manufacturing, accounting, etc.); but instead most personnel work in interdisciplinary teams organized around products [Boudette 90].

Quality circles (QCs) provide an interesting case of team organization because they are perceived by some to have been less than completely successful. *The Economist* [Econ 92] reports that the only U.S. firm to win the Japanese Deming Award, Florida Power and Light Company, has disbanded most of its quality teams. Hyde [Hyde 86] provides a list too lengthy to reproduce here of reasons why the use of QCs and similar teams has waned. His conclusions are that QCs were oversold to organizations that did not understand and were not ready for true team organization, and whose management ultimately lacked conviction in the team building process.

10.5 Suggestions for Introduction and Use

Suggestions for use are specific to the categories of IGMs and, sometimes, to methods within the categories.

10.5.1 Group Dynamics Facilitation

In general, great attention should be paid to the task being undertaken. If the task is nonstandard (i.e., not requirements analysis or something like it), then care should be exercised in considering the suitability of the task to the method. As mentioned earlier, the task should be one that lends itself to division into parts.

Corbin [Corbin 91] provides the following recommendations specifically for JAD, although they are clearly more generally applicable.

- Team members need to be those who "can't be spared from their current jobs." The task will be performed in the best manner possible only with the participation of those people whose knowledge of the application is so critical that they and others feel that they cannot leave it alone for a week or two.

- Get management support at a level high enough to free critical individuals to participate.

- Choose facilitators carefully; a good facilitator will have excellent people skills and a good command of systems development concepts. This is a rare combination, even among consultants in this business.

- Meet off-site where participants cannot be called away for phone calls or crises.

- The Information Systems (IS) staff should be seen, not heard; the team should be composed of mostly users, and the developer's role is to listen and provide answers to technical questions.

In using group dynamics methods, one should realize that the specific procedures and rules have very important behavioral justifications, even when they are not apparent. Some methods described elsewhere in this report, such as formal inspection, use group dynamics techniques to facilitate their implementation, and there is a lesson that can be drawn from this observation. The success of formal inspections has been shown to be quite sensitive to variation in specific rules of conduct. Review methods that depart from the Fagan rules usually are not as effective, because they allow undesirable side-effects of group activity to creep back into the task. Similarly, group dynamics facilitation of all kinds are more likely to be successful when the rules of conduct are obeyed. If the decision-making task cannot be usefully accomplished within the rules, then perhaps the method is not right for the task.

10.5.2 Groupware Methods

Little can be said here because of the tremendous variation across groupware and associated methods. Again, suitability of the method to the task should be important. Also, those who use these methods should remain aware that the technology is likely to be popular whether or not it is effective. For this reason, those who implement groupware methods should insist on evidence of increased effectiveness to offset the costs of the associated technology. On the other hand, there is some evidence that such technologies may become useful in ways that were not anticipated. Implementers should be alert for "second-order" benefits of this sort, that arise from people finding the technology useful in ways that were not planned. As with many things, simple is probably better in groupware methods.

10.5.3 Team Organization

Some lessons can be drawn from instances of success and from Hyde's [Hyde 86] analysis of the failure of QCs.

- It is often useful to group people from different functional areas together and associate them with a project or an objective to be accomplished.

- It is important that the group perceives itself as a team. An on-paper-only reorganization accomplishes nothing; team feeling is built through frequent interaction, collocation, and related interpersonal factors.

- The team environment should be designed to maximally facilitate informal communication. Team organization is an attempt to substitute informal modes of communication and group structure for more formal ones; thus, attention must be paid to making the informal modes possible.

- It should be recognized that the transition from a formal, hierarchical organization to a flatter, less formal, team-based organization is dramatic. Time and attention should be paid to who the perceived losers are in the reorganization; if middle management, for example, perceives a threat to its power base, they can subvert team organization.

- Team organization must really be team organization, not traditional organization disguised; to achieve the sort of informal control required in team organization, there must be two-way commitment, team members to the effort and the organization to the team members.

- Team organization is not a few specialized methods like Pareto analysis, or cause and effect diagramming; these tools may be used by the team, but they are not what defines the team.

10.6 How Interdisciplinary Group Methods Are Related to the Capability Maturity Model

The various interdisciplinary group methods described correlate to the level 3 key process area, intergroup coordination. The methods described provide assistance to organizations wishing to better coordinate the activities and interaction among various functions involved with the development of software products. These functions can include groups within software engineering (e.g., a design team) or to groups external to software engineering (e.g., hardware engineering, marketing, service, test).

10.7 Summary Comments on Interdisciplinary Group Methods

IGMs seem to hold considerable promise for improving the way software is developed, but only if great attention is paid to the appropriateness of the method to the task. There is a tendency with these methods to think that they will solve too many problems. In areas where they are tried and true, like requirements definition, no organization should hesitate to use them, provided the organization can get the level of management support necessary to assemble the right participants. Also, for those who have some behavioral sophistication and are willing to be pioneers, combining methods from the three categories may hold great promise.

10.8 References and Further Readings - IGMs

[Boudette 90] Boudette, N. E., "Give Me a 'T!' Give Me an 'E!'...," *Industry Week*, 62-65, January 8, 1990.

[Corbin 91] Corbin, D. S., "Team Requirements Definition: Looking For A Mouse and Finding an Elephant," *Journal of Systems Management*, 28-30, May, 1991.

[Davis 69] Davis, J. H., *Group Performance*, Addison-Wesley, Reading, MA, 1969.

[Delbecq 75] Delbecq, A. L., Van de Ven, A. H., Gustafson, D. H., *Group Techniques for Program Planning*, Scott, Foresman, Glenview, IL, 1975.

[Dennis 88] Dennis, A. R., George, J. F., Jessup, L. M., Nunamaker, J. F., Vogel, "Information Technology to Support Electronic Meetings," *MIS Quarterly*, Vol. 12, No. 4, 591-624, 1988.

[Desanctis 87] Desanctis, G. and Gallupe, R., "A Foundation for the Study of Group Decision Support Systems," *Management Science*, Vol. 33, No. 5, 589-609, 1987.

[Dumaine 89] Dumaine, B., "How Managers Can Succeed Through Speed," *Fortune*, Feb. 13, 1989.

[Econ 92] "Cracks in Quality," *The Economist*, 67-68, April 18, 1992.

[Gladstein 84] Gladstein, D. L., "Groups in Context: A Model of Task Group Effectiveness," *Administrative Science Quarterly*, Vol. 29, 499-517, 1984.

[Gordon 89] Gordon, F., Isenhour, R., "Simultaneous Engineering," *Engineering Manager*, Jan. 30, 1989.

[Gronbaek 88] Gronbaek, K., "Rapid Prototyping with Fourth Generation Systems," Technical Report, Aarhus University Computer Science Department, Aarhus, Denmark, November, 1988.

[Hegedus 86] Hegedus, D. M., Rasmussen, R. V., "Task Effectiveness and Interaction Process of a Modified Nominal Group Technique in Solving an Evaluation Problem," *Journal of Management*, Vol. 12, No. 4, 545-560, 1986.

[Hyde 86] Hyde, W. D., "How Small Groups Can Solve Problems and Reduce Costs," *Industrial Engineering*, 42-49, December, 1986.

[Janis 72] Janis, I. L., *Victims of Groupthink*, Houghton Mifflin, Boston, MA, 1972.

[Katterman 90] Katterman, L., "The Future of Collaboration," *Research News*, The University of Michigan, Division of Research Development and Administration, May-June, 1990.

[Liker 93] Liker, J. K., Kamath, R. R., Wasti, N., Nagamachi, M., "Supplier Involvement in Product Development in Japan and the U.S.," University of Michigan working paper, 1993.

[Nunamaker 88] Nunamaker, J., Applegate, L., Konsynski, B., "Computer-Aided Deliberation: Model Management and Group Decision Support," *Operations Research*, Vol. 36, No. 6, 826-848, 1988.

[Opper 88] Opper, S., "Making the Right Moves with Groupware," *Personal Computing*, 134-140, December, 1988.

[Ouchi 81] Ouchi, W. G., *Theory Z*, Addison-Wesley, Reading, MA, 1981.

[Schmelzer 89] Schmelzer, H.J., "How to Gain a Competitive Edge", *Siemens Review*, Nov./Dec. 1989, Vol. 56, 6/89.

[Sproull 86] Sproull, L., Kiesler, S., "Reducing Social Context Cues: Electronic Mail in Organizational Communication," *Management Science*, Vol. 32, 1492-1512, 1986.

[Taff 91] Taff, L. M., Borchering, J. W., Hudgins, Jr., W. R., "Estimeetings: Development Estimates and a Front-End Process for a Large Project," *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, 839-849, August, 1991.

[Zigurs 88] Zigurs, I., Poole, M. S., DeSanctis, G., "A Study of Influence in Computer-Mediated Group Decision Making," *MIS Quarterly*, Vol. 12, No. 4, 625-644, 1988.

11 Software Reliability Engineering (SRE)

11.1 Overview

Software reliability engineering is a statistical technique for predicting failure rates for a software system before it is released. This information is valuable for a number of reasons.

- Since the failure rates decrease with testing and debugging, predictions of failure rates can help developers know when to stop testing (i.e., when the quality of the software is adequate).
- Having a prediction of failure rates makes decisions about tradeoffs between performance, cost, schedule, reliability, and other factors easier and more explicit.
- Knowing predicted failure rates associated with various development methods can help refine the development process.
- Accurate prediction of failure rates makes it possible for producers of software to guarantee to their customers failure rates below certain tolerances.

The overall goal of SRE is to improve the way in which software quality can be managed – to cast what was formerly a matter of intuition and guesswork into more objective terms.

11.2 History/Background

Software reliability started in the early 70's with the publication of a series of papers in the academic literature. Musa [Musa 75] heightened interest in the area by demonstrating that reasonably accurate predictions could be made using reliability models. Since then, a tremendous variety of models have proliferated, each with their own advantages and drawbacks. (See [Abdel-Ghaly 86] for a partial survey.) Because software reliability has arisen in the research laboratory where it has been tested under rigorous conditions, there is high confidence in the claims made in the literature. Based on these claims, SRE would seem to hold great promise for improving the way software is developed. Also because of its research ancestry, however, and because it is a mathematically challenging subject, there are significant challenges yet to be met in adapting SRE for common use by practitioners.

11.3 Description of Software Reliability Engineering

A *software failure* is "a departure of a computer program's operation from the user requirements; a failure may be a 'crash', in which a system ceases to function, or a simpler malfunction, such as the display of an incorrect character on a monitor" (Musa, 1989). A failure is distinct from a fault, which is the specific software feature that caused the failure (e.g., a coding error). The end result of SRE is to produce predictions of failure rates as expressed in a number of closely related ways. Examples of predicted failure rates include:

- Failure intensity is the number of failures of a software system occurring in a standard time period; "2.5 failures per 1000 hours of operation" is a statement of failure intensity.
- Reliability is the probability that a software system will operate failure-free for a standard time interval; "0.95 for 24 hours" is a statement of system reliability.
- Mean time to failure, often abbreviated MTTF (or, equivalently, mean time before failure, or MTBF), is the average duration of system operation expected before the occurrence of the first failure; "1000 hours MTTF," is a statement of mean time to failure.

Reliability predictions are routinely used in the design and manufacture of hardware components to determine how often parts and components will require service. Such predictions are critical to customer satisfaction. Software reliability, while also critical to customer satisfaction, is different from hardware reliability in that failure is not due to a "wearing out" process [Musa 75]. Software failures occur when the system is exposed to an input for which it was not designed. Because of the relative logical complexity of software, its large number of possible states and inputs, it is usually impossible to test software comprehensively. Software reliability is, then, not so much a statement about the durability of the components of a system as a statement of the level of confidence in the design. By testing and debugging, developers can grow more confident in the design of a system. The central advantage of SRE is that it provides a structured means of determining quantitatively how much confidence in the design of a system is warranted, at any point in time.

The actual models involved in SRE are mathematically complex, but some have been automated for applied use [Musa 89]. As testing is carried out, the rate at which failures are discovered provides information about the number of faults in the system. As failures are discovered and repaired in testing, failure intensity decreases, and reliability and MTTF increase. They increase in a manner that can be described mathematically; thus, at any point in time thereafter, a quantitative estimate of system reliability can be produced.

As with any statistical technique, predictive accuracy depends greatly on estimates of model parameters, which in turn depend on the data used to estimate the parameters. Some parameters are quite constant over time and across environments, but others are not. This means that SRE models, like estimation models, may vary in their performance across environments. That is, SRE models must be "calibrated" to particular environments and some models may be better than others for specific environments. Parameter estimation depends also on accurate gathering and recording of data concerning the timing of fault discovery, system characteristics, etc.

An additional benefit of SRE is that it substantially enhances the level of developer-customer dialog [Musa 87]. Customers and developers must reach agreement on what constitutes a "failure" in their environment, and this usually leads to better specification of requirements. Reliability figures can easily be related to the operational costs of failures and development costs, providing customers with a basis for making decisions about tradeoffs.

11.4 Experience with SRE

Software reliability engineering is anomalous among the methods in this report. Unlike many methods for which enthusiasm abounds, SRE has been rigorously tested and found to be effective. Despite the strong evidence of its benefits, however, SRE is not widely deployed, compared to many of the methods in this report. A 1988 survey revealed that only four percent of respondents reported using SRE [Karcich 91]. There are at least two reasons for this. First, SRE is relatively new, although no newer than many of the methods that have garnered large followings. Second, and most significant, SRE comes out of the laboratory looking like research. It has not yet been effectively packaged in a way that makes it accessible to many development organizations. This leads to the following problems:

1. Papers on reliability are usually deep into ominous looking equations by page three or so.

2. Software reliability researchers are more inclined to make minor corrective changes to their models than to confront issues relevant to practice [Levendel 91]. As a result, many practitioners have been put off.

Not all practitioners have been put off, however. Some have been attracted by the findings in the laboratory that predicted failure intensities within 15 percent or so can be achieved [Musa 75]. Musa [Musa 89] reports the following instances of model implementations:

- Programs that implement reliability models are now available as part of at least two commercial products.
- The U.S. Naval Surface Weapons Center in Dahlgren, Virginia, has implemented several models in a program that runs under MS-DOS.
- Hewlett-Packard employed a reliability model to determine when to release the firmware for their Frontier series of remote terminals. They reported that after one year of checking the predictions, slightly fewer than the predicted number of failures had occurred.
- Hewlett-Packard also used reliability models to estimate expected failure intensities for firmware in two other terminals after they were released to users. Predictions were within 12 percent of actuals.
- Designers within a 1986 AT&T project used a reliability model to predict completion dates – dates by which failure rates would reach target levels. After two of the ten weeks of testing were completed, the model predicted the completion date to within one week.
- AT&T used reliability models to predict the rate of field maintenance requests for its 5ESS switching system. Predictions differed from actuals by 5 to 13 percent.

Other authors cite examples of uses in which predictions are acceptably close to actuals [Shooman 87], [Everett 87], [Ejzak 87], [Drake 87].

SRE has been used successfully to ensure high levels of reliability in safety-critical situations. Stark [Stark 87] recounts how SRE has been used successfully in the development of NASA's Shuttle Mission Training Facility, to notify management of changes in the reliability of the system due to new requirements implemented, to understand which functions of the simulation system are the reliability problem areas, and to identify trends in the reliability of the software over time. The use of SRE on the space shuttle project suggests possible future uses in certifying the reliability of code that cannot be allowed to fail. Recent disasters related to software failures include the software malfunction of the Therac-25, a device for destroying tumors, which killed several people before the problem was found [Leveson 93], and several aircraft and air traffic control mishaps [Lee 91]. Theorists warn that applications which demand ultrahigh levels of reliability are not suited to existing models [Musa 89]. Keller [Keller 91], however, reports just such an application of SRE, on the space shuttle's "primary avionics software system."

11.5 Suggestions for Introduction and Use

Because SRE has been focused primarily on research, there is minimal "how-to" advice in the literature. Some limited advice is offered by SRE pioneers:

- Training is an absolute necessity for users of SRE [Musa 89]. Although users need not become expert in the intimate details of the models, they should have a thematic understanding of the statistical issues involved.
- Model selection is important, as is calibrating the chosen model to the development environment [Abdel-Ghaly 86]. Adams [Adams 91] has reported that common reliability models have been unsuccessful at Cray Research, because their development processes fail to conform to model assumptions. This fact emphasizes the importance of understanding the models' basic assumptions and determining whether your own processes adequately approximate the assumptions.

11.6 How SRE Is Related to the Capability Maturity Model

SRE most closely correlates to the level 4 key process area, quantitative process management. SRE provides models and tools for applying more sophisticated measures. Since predictions of software reliability can be made using SRE, the predicted data can also be used for tracking progress during development.

11.7 Summary Comments on SRE

Software reliability engineering seems to hold great promise as a means of improving the way software is developed. Unfortunately, it has been slow to emerge from the research laboratory. The gap between researchers and practitioners has yet to be fully closed. For those willing to attempt to bridge that gap, the benefits may be considerable. Of particular benefit to software developers is the possibility of using SRE to improve software reliability. When the reliability of a software system can be predicted, the data collected through the development process can be used for controlling and improving the process.

11.8 References and Further Readings - SRE

[Abdel-Ghaly 86] Abdel-Ghaly, A.A., Chan, P. Y., Littlewood, B. "Evaluation of Competing Software Reliability Predictions," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, September, 1986.

[Currit 86] Currit, P. A., Dyer, M., Mills, H. D., "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, January, 1986.

[Drake 87] Drake, D., "Reliability Theory Applied to Software Testing," *Proceedings of the Fall Joint Computer Conference*, IEEE, 1987, Dallas, Texas.

[Ejzak 87] Ejzak, R. P., "On the Successful Application of Software Reliability Modeling," *Proceedings of the Fall Joint Computer Conference*, IEEE, 1987, Dallas, Texas.

[Everett 87] Everett, W. W., "Software Reliability Applied to Computer-Based Network Operation Systems," *Proceedings of the Fall Joint Computer Conference*, IEEE, 1987, Dallas, Texas.

[Hecht 86] Hecht, H., Hecht, M., "Software Reliability in the System Context," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, January, 1986.

[Karcich 91] Karcich, R. M., "Panel Discussion: Practical Applications of Software Reliability Models," *Proceedings of the 1991 International Symposium on Software Reliability Engineering*, IEEE Computer Press, 1991.

[Keller 91] Keller, T., "Panel Discussion: Practical Applications of Software Reliability Models," *Proceedings of the 1991 International Symposium on Software Reliability Engineering*, IEEE Computer Press, 1991.

[Lee 91] Lee, Leonard, *The Day the Phones Stopped*, Donald I. Fine, Inc., NY, 1991.

[Levendel 91] Levendel, Y. H., "Panel Discussion: Practical Applications of Software Reliability Models," *Proceedings of the 1991 International Symposium on Software Reliability Engineering*, IEEE Computer Press, 1991.

[Leveson 93] Leveson, N.G., Turner, C.S., "An Investigation of the Therac-25 Accidents," *IEEE Computer*, July, 1993.

[Musa 75] Musa, J. D., "A Theory of Software Reliability and Its Applications," *IEEE Transactions on Software Engineering*, SE-1, No. 3, September, 1975.

[Musa 87] Musa, J. D., Iannino, A., Okumoto, K., *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, 1987.

[Musa 89] Musa, J. D., "Tools for Measuring Software Reliability," *IEEE Spectrum*, February, 1989.

[Shooman 87] Shooman, M. L., "Yes, Software Reliability Can Be Measured and Predicted," *Proceedings of the Fall Joint Computer Conference*, IEEE, 1987, Dallas, Texas.

[Stark 87] Stark, G. E., "Monitoring Software Reliability in the Shuttle Mission Simulator," *Proceedings of the Fall Joint Computer Conference*, IEEE, 1987, Dallas, Texas.

12 Quality Function Deployment (QFD)

12.1 Overview

Quality function deployment, usually called simply QFD, is a way of structuring the translation of customer requirements into product features. The method has evolved to fit different areas where it has been applied. Thus QFD means different things to different people. However, all of its forms share reliance on a matrix diagram, called "the house of quality." This diagram is used to:

- Document, group, and prioritize customer requirements.
- Relate requirements to engineering factors.
- Highlight interactions among engineering factors.

While there is no single "QFD process," some software organizations have embedded QFD into their development cycle. The purpose of QFD is to structure and focus early stages of design so that the customers needs will be better served by the final product.

12.2 History/Background

QFD originated in 1972, at Mitsubishi's Kobe shipyard, and its use spread almost immediately [King 87]. Toyota and its suppliers added many of the features that are part of the modern QFD approach [Hauser 88]. In Japan and the U.S., QFD has been widely and successfully used in the auto industry. QFD is, however, relatively new to software development. Yoshizawa et al. ([Yoshizawa 90a], [Yoshizawa 90b]) recount application on products with software components in Japan, and Zultner [Zultner 92] reports that organizations such as AT&T Bell Laboratories, Digital Equipment Corporation (DEC), Hewlett-Packard (HP), IBM, and Texas Instruments (TI), have begun to use QFD to design software products. But the quantity of documented, publicly available evidence of its successful use is relatively small. Nevertheless, QFD has been successful in a wide variety of nonsoftware applications, and most software organizations do want to improve their requirements gathering and design processes. These two facts suggest that QFD might offer benefits to software developers willing to pioneer in its use.

12.3 Description of QFD

QFD has many complex variations. Rather than attempt to describe all of its variations, we recount the core elements of QFD as it was originally conceived for generic products (i.e., not necessarily software products). This description is drawn primarily from [Hauser 88]. We then describe some common variations, featuring those that have occurred in software development settings.

1. QFD begins by finding out what customers say they want. Customers are asked what they like about a product of this type (e.g., "what makes a good car door?"). In software development, this aspect of QFD requires some modification, since product types are harder to define and there is (usually) no mass-market from which to draw customers experienced with a range of such products. But software organizations do have procedures for eliciting requirements from their users, which QFD can be adapted to incorporate. In some existing software applications of QFD, its definition has been effectively expanded to include many specific methods for deriving requirements, such as joint application design (JAD) and other group meeting and facilitation techniques [Cohen 88], [Brown 91]. Two classes of requirements are not supplied by customers but are particularly important for software products. They are:

    a. "expected quality" requirements, which customers do not mention because they assume such requirements will be met by any product of this type; and,

    b. "exciting quality" requirements, which represent features that are new, that the customer has never encountered but will like, made possible by technological advances or design innovations.

2. Requirements of all types are listed, where possible in the customer's own words, and grouped into categories.

3. Weights are supplied for each requirement, to indicate their relative importance. To the degree possible, weightings come directly from the customer. Weights are written along side each requirement, usually in terms of percentages which total 100.

4. Next, the current product is compared to competitors' products for each requirement. For example, adjacent to the requirements list might be a scale on which the product is rated from one to five, along with similar ratings for competitors' products. Comparative rating helps highlight areas where the product is particularly strong or weak, relative to the competition, thus pointing out either advantages to be exploited or the need for corrective action. Clearly this aspect of QFD will require modification in many software development environments, where it is

more difficult to determine comparable products and find customers capable of making product comparisons. In some environments, where software is produced exclusively for in-house use, discussion of competitors may not be appropriate; hence this part of the QFD matrix may be omitted entirely.

5. Once requirements are listed and weighted, the engineering factors that can affect the requirements are themselves listed, across the top of the matrix. Engineering factors are characteristics of the product that are not directly experienced by the customer. In a typical automotive example, an engineering factor might be the strength of a windshield wiper spring and the requirement that it affects might be the incidence of windshield streaking in a rain storm. In software development, these factors are more likely described by technical characteristics of the system, such as the way data are organized in a database. The direction and strength of influence (on a scale, say, from -10 to 10) of engineering factors on requirements is represented by filling in the matrix. In some applications, the weighting of a requirement is multiplied by this influence number, to help highlight opportunities for improvement; where this is done, the product of the two numbers is also written in the body of the matrix.

6. Usually, engineering factors are not completely independent of each other. Using thicker steel may make a car door more crash resistent, but it may also increase the force needed to swing the door open and the weight of the car, thus reducing gas mileage. In software, changing the way the data are organized, say choosing to store a data element in one place rather than two, may enhance the integrity of the data, but may adversely affect processing time. These interactions of engineering factors are represented in a triangle above the list of engineering factors. By tracing along the appropriate axes of the triangle from any two engineering factors, one comes to a joint coordinate wherein the interaction's strength and direction may be indicated. This part of QFD is extremely valuable in that it makes explicit the sort of interactions that lead to unanticipated consequences of design changes.

7. Finally, the bottom of the matrix may be used to further quantify engineering factors, to list cost of change estimates, and to set targets for engineering factors. It is customary in QFD to list single value targets for engineering factors, rather than tolerances, because tolerances lead to production of the lowest level within tolerance. Figure 12-1 depicts an example "house of quality" diagram.

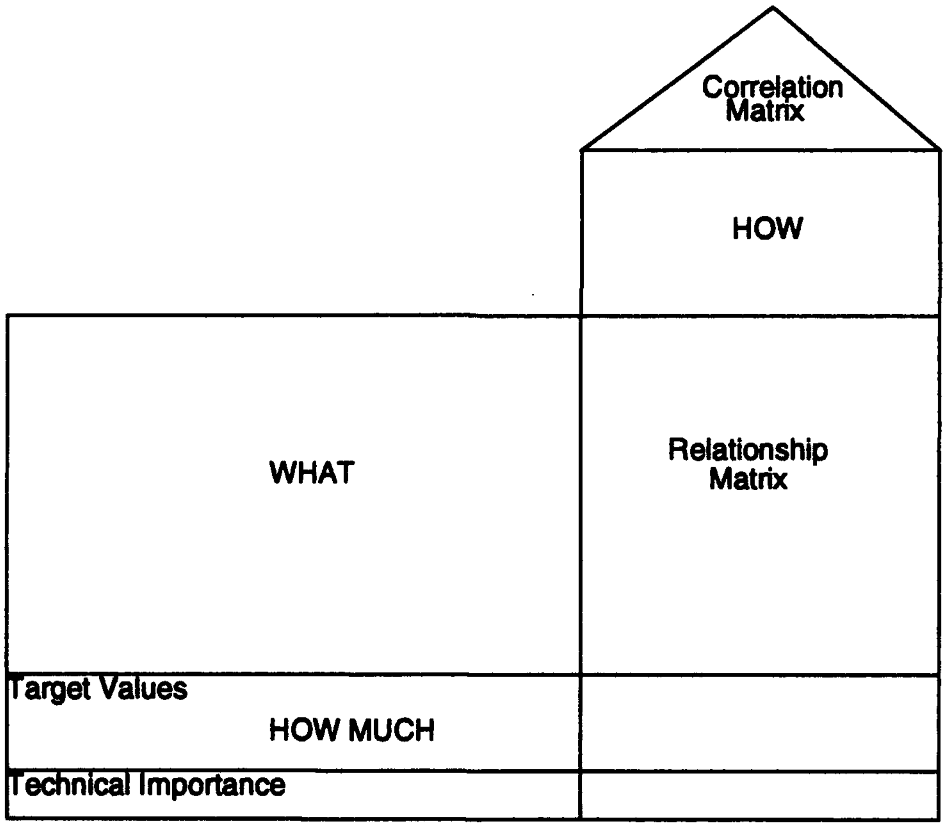

Figure 12-1. "House of Quality" Diagram

QFD is often used in a hierarchical manner. A first house of quality might have customer requirements related to engineering factors. A second house might have engineering characteristics related to component characteristics, and so on, down to the lowest level of implementation detail. In software development, the house and each hierarchical level might correspond to a stage of the development cycle.

What few accounts of QFD's use in software development exist tend to emphasize processes for eliciting requirements and constructing the matrix. Brown [Brown 91] recounts how QFD is used at AT&T Bell Laboratories by establishing a cross-functional "QFD team" which then follows a step-by-step process to construct the matrix (or matrices). A variety of group facilitation techniques not unique to QFD are emphasized by Cohen [Cohen 88] who provides an account of the use of QFD at DEC. Zultner ([Zultner 90], [Zultner 91], [Zultner 92]) promotes a complex array of QFD-like matrices, used, for example, to identify different customer segments and relate them to requirements. One often claimed benefit of QFD is that, by serving as a central communication device, it facilitates simultaneous engineering of multiple product components, thereby shortening cycle times.

12.4 Experience with QFD

As has been noted, the volume of publicly available documentation on use of QFD in software development is small compared to what is available for the process improvement methods presented earlier. Brown [Brown 91] reports successful use within AT&T on already completed and ongoing products with software components. In areas where QFD has been used, Brown claims that:

- Customer needs were generally better understood.
- More critical needs were met.
- Credibility with the customer was enhanced.
- Unmet customer needs were discovered.
- Communications were generally better among members of the design team.

Cohen [Cohen 88] recounts the following benefits from using QFD at DEC:

- QFD is more efficient than other product planning processes; in the same amount of time a more detailed statement of customer needs can be derived.

- The structure inherent in construction of the house of quality makes product planning easier to carry out.
- QFD makes explicit detailed information on interrelationships between product features and customer needs.
- QFD provides a tool for identifying disagreement and converting that into consensus.
- The house of quality acts as an archive of the planning process; trade-off decisions are well documented.

It is unclear, however, from Cohen's account how much software content was included in products designed by using QFD.

Yoshizawa et al. indicate that an effort to extend QFD to software development has been underway in Japan since the late 1970s [Yoshizawa 90a], [Yoshizawa 90b]. They report, as well, that several companies in Japan have been applying QFD to software development since 1982. The companies applying QFD include CSK, IBM Japan, NEC IC Micon, and Nippon Systems. Unfortunately, publicly available English language accounts of these applications are difficult to obtain.

12.5 Suggestions for Introduction and Use

It is clear from the literature on QFD implementations that QFD is almost always tailored significantly to the application at hand. King [King 87] notes that successful tailoring usually brings the tool to bear on "key interfaces," places in the design process where communication is difficult but essential to the eventual performance of the product. Brown suggests that organizations trying QFD for the first time should get help from someone experienced with the technique. He also emphasizes the importance of training, including training of management personnel who must understand the rationale, time requirements, and expected benefits of QFD. Cohen [Cohen 88] provides the following suggestions for those seeking to minimize the substantial time required to carry out QFD:

- Break QFD participants into subgroups, each responsible for developing a different part of the matrix. In doing so, however, the multidisciplinary nature of the original group should be preserved in the subgroups. When subgroups reassemble into a larger group, provide reports from each subgroup to make sure subgroups all share a common understanding.

- Construct a large (three to five foot tall) house of quality with removable "shingles." Groups can take shingles, fill them in, and return for another shingle.

- Construct the matrix only for a subset of most important customer needs; alternatively, fill in the body of the matrix only for categories of requirements, rather than for every specific requirement.

Obviously there are risks in any abbreviation of customers needs. But these risks, along with the other important issues involved in making QFD usable in a specific environment, can be addressed. The best uses of QFD for software will probably, judging from the literature on QFD, occur in situations where QFD is carefully and intelligently customized to be appropriate; failures will likely occur where the method is adopted by rote.

12.6 How QFD Is Related to the Capability Maturity Model

QFD is a structured means of transforming customer requirements into design specifications. It is identified as an example method for tracing and prioritizing software quality needs of the organization, customer, and end user within the CMM KPA software quality management. The KPA software quality management is within level 4, the managed level of the CMM. QFD also could be used to improve the performance of an organization to do requirements management, which is a level 2 (repeatable) KPA.

12.7 Summary Comments on QFD

Software development is often negatively impacted by complex interactions between design features. Small design changes ripple through projects producing consequences that no one anticipated. While software developers have evolved structured means of doing their jobs, identification of interactions is not an explicit emphasis of most software development methodologies. QFD – a structured way of identifying and understanding interactions – seems poised, then, to offer this very significant advantage to software developers. This advantage, added to QFD's potential for improving developer's focus on customer requirements, makes the method seem very promising.

12.8 References and Further Readings - QFD

[Brown 91] Brown, Patrick, "QFD: Echoing the Voice of the Customer," *AT&T Technical Journal*, March-April, 1991.

[Cohen 88] Cohen, Louis, "Quality Function Deployment: An Application Perspective form Digital Equipment Corporation," *National Productivity Review*, Summer, 1988.

[Hauser 88] Hauser, John R., and Clausing, D. "The House of Quality," *Harvard Business Review*, May-June, 1988.

[King 87] King, Robert. "Listening to the Voice of the Customer: Using the Quality Function Deployment System," *National Productivity Review*, Summer, 1987.

[Thackeray 89] Thackeray, R., Van Treeck, G., "Quality Function Deployment for Embedded Systems and Software Product Development" *GOAL/QPC Sixth Annual Conference Proceedings*, Boston, 1989.

[Yoshizawa 90a] Yoshizawa, T., Akao, Y., Ono, M., Sindou, H. "Recent Aspects of QFD in the Japanese Software Industry," *ASQC Quality Congress Transactions* - San Francisco, 1990.

[Yoshizawa 90b] Yoshizawa, T., Togari, H., Kuribayashi, T., and the CSK Software Quality Assurance Committee, "Quality Function Deployment for Software Development," in *Quality Function Deployment: Integrating Customer Requirements into Product Design*, ed. Yoji Akao, translated by Glenn Mazur, Productivity Press, Cambridge, MA, 1990

[Zultner 90] Zultner, Richard E. "Software Quality Deployment: Applying QFD to Software," *2nd Symposium on QFD Transactions*, Novi, Michigan, June 1990.

[Zultner 91] Zultner, Richard E. "Before the House: The Voices of the Customers in QFD," *3rd Symposium on QFD Transactions*, Novi, Michigan, June 1991.

[Zultner 92] Zultner, Richard E. "Quality Function Deployment (QFD) for Software", *American Programmer*, February, 1992.

13 Total Quality Management (TQM)

13.1 Overview

Total quality management, usually denoted by its acronym, TQM, refers to a philosophy of management and a collection of methods with the central goal of improving the quality of an organization's products and the satisfaction of its customers. TQM has become pervasive throughout U.S. companies; not since the scientific management movement in the early years of this century has a change in management practices been so embraced by the business community at large [Ross 93]. One consequence of its popularity is that TQM has become somewhat loosely defined. It is common to use TQM to describe nearly any method intended to improve processes or promote quality in products. However, the creation in 1987 of the Malcolm Baldrige National Quality Improvement Award, given by the U.S. Department of Commerce to companies that excel in quality management, has restored some coherence to the TQM concept. The legislation that created the award explicitly called for creation of guidelines and criteria that companies could use to evaluate their quality improvement efforts [Garvin 91]. By many accounts, these guidelines and criteria have come to define TQM.

13.2 History/Background

TQM arose from the quality management techniques employed by Japanese manufacturers, which are commonly considered responsible for the quality superiority of many Japanese products. Ironically, Americans W. Edward Deming and J. M. Juran are credited with developing many of the techniques used so effectively to seize U.S. market share from American firms. In fact, the Japanese national prize for quality is named for Deming; the "Deming Prize" was the original inspiration for the Baldrige Award. More recently, TQM was derived from the philosophy and techniques expounded by Kaoru Ishikawa [Ishikawa 85] under the label "total quality control" or "TQC." TQM is at use in many software organizations, to a greater or lesser extent, but it is important to realize that TQM is an organizational undertaking; that is, it must be broadly deployed, not just in software development. Thus, for TQM to be truly "total," it must be adopted not only in software development, but also as part of a larger organizational effort.

13.3 Description of TQM

TQM is used to describe such a vast array of practices that it would be impossible to detail them all in the space provided. However, there are certain common features in most TQM efforts. Those common features are captured in the criteria for the Baldrige

Award. The scoring framework for the Baldrige Award therefore provides a structure for describing TQM. Note that we present TQM here in its generic sense; i.e., not necessarily as adapted to software development. Software development variations are discussed at the end of this section. Much of what follows is derived from [Garvin 91] and [Ross 93].

There are seven categories of management practice that together constitute TQM. They are:

- Leadership.
- Information and analysis.
- Strategic quality planning.
- Human resources utilization.
- Quality assurance of products and services.
- Quality results.
- Customer satisfaction.

Leadership reflects the degree to which the organization's management has internalized quality objectives and values. *Information and analysis* concerns how effectively the organization gathers and analyzes information concerning its own quality performance, especially as compared to its competition. *Strategic quality planning* refers to the degree to which quality concerns are integrated into the organization's planning process, and how quality plans and objectives are communicated to operating units. *Human resource utilization* concerns whether the organization has succeeded in promoting a culture of quality in all of its workers and also the degree to which employees are "empowered"—authorized to take action to make changes that improve quality. *Quality assurance of products and services* examines actual implementation of quality principles, tools, and techniques, including design-for-manufacture, just in time (JIT), continuous process improvement, quality assessment, statistical process control (SPC), etc. *Quality results* is what it sounds like—demonstrable improvement in product quality over time. And finally, *customer satisfaction*—considered the most important category—focuses on the organization's relationships with its customers: customer needs determination, customer service capability, demonstrable improvements in customer satisfaction, etc.

13.4 TQM Tools

There are also seven specific tools that are considered standard for TQM [Zultner 88]. The tools themselves are not mandatory in an organization that adopts TQM, but fulfillment of the tools' respective purposes is required. The tools are:

- Check sheets useful for tabulating errors for each work product.
- Graphs such as trend charts, which present defect data over time.
- Cause and effect diagrams, which show all possible causes of some condition.
- Pareto charts, which rank types of defects by frequency.
- Histograms, which are bar charts for displaying frequency data.
- Scatter diagrams, which are plots that relate two measurable quantities to each other and therefore permit discovery of relationships between the quantities.
- Control charts, which show defects plotted over time with statistical control limits.

These tools are often used in a structured sequence by interdisciplinary teams, called quality circles (see description in Chapter 10), at all levels of the organization.

A central theme in TQM is identification and improvement of production processes. Such processes frequently cross boundaries between suborganizations within the overall organization; thus, process stages are subject to differing influences. Different parts of the process are "owned" by different members of the organization. However, for successful production of quality products, it is essential that processes operate smoothly across boundaries. TQM emphasizes the identification of essential processes and the establishment of ownership for the overall processes. If there is one "process owner," it will be more likely that each element of the process will be integrated successfully into the whole.

Because of its emphasis on essential processes, TQM focuses on rewarding excellence in process rather than end results [Shaffer 92]. The rationale behind this focus is that end results are usually influenced by factors outside the control of those seeking to improve quality. Therefore, favorable results may arise from faulty processes and vice-versa. However, the most consistently favorable trends will result, it is argued, from focus on factors that can be controlled. Quality teams within organizations are, therefore, often rewarded for excellence in adhering to an improvement process, rather than for the actual results of their actions. Similarly, the largest part of the Baldrig‿ Award scoring is on process, not results. Not all commentators approve of this de-emphasis of results [Shaffer 92].

In fact, the way TQM has been implemented in many U.S. companies, especially in efforts to win the Baldrige Award, has become a subject of acute controversy. In particular, a pitched debate rages around the Baldrige Award itself; surprisingly, some of the most vocal critics are the founders of the quality movement. Deming [Deming 92] himself has written: "The Baldrige Award does not codify the principles of management of quality. It contains nothing about quality." And of an article explaining and defending the Baldrige Award written by a member of the award's board of overseers, Deming [Deming 92] has said "It transgresses all that I try to teach." Crosby [Crosby 92], author of the influential book *Quality is Free*, argues that "the Baldrige criteria have trivialized the quality crusade perhaps beyond help." His condemnation is unequivocal: "One day this do-it-yourself kit may be recognized as the cause of a permanent decline in product and service quality management in the United States."

Specific criticisms of the Baldrige Award and common implementations of TQM are varied and can be found in the January-February 1992 issue of *The Harvard Business Review*. We recommend that the issues raised there be seriously considered by anyone thinking of bringing TQM to their own organization. One central issue worth mentioning here involves the degree to which quality principles and methods can be tailored while still retaining their value. Founders of the quality movement like Deming and Crosby, who are unhappy with the Baldrige criteria, argue that many implementations of quality assurance methods transgress crucial principles of quality assurance. They note further that the Baldrige Award, in taking an "ecumenical approach" [Garvin 92], fails to penalize these transgressions, thereby encouraging faulty practice. Advocates of the Baldrige Award (e.g., [Garvin 91]) reply by denying that the principles of Deming and Crosby are fundamental, and arguing that there are many roads to quality.

13.5 Experience with TQM

Many U.S. companies have experience with TQM, but in most companies TQM has been in place for only a short time. This makes it difficult to assess the long-term effectiveness of TQM. The most significant evidence to date that TQM improves organizational performance comes from a May 1990 General Accounting Office (GAO) report. The GAO surveyed and interviewed 20 companies that scored well in either the 1988 or 1989 Baldrige competition. They concluded that there was a cause-and-effect relationship between adherence to TQM practices embodied in the Baldrige criteria and organizational performance, measured by employee relations, productivity, customer satisfaction, or profitability. They also found that there was no "cookbook" approach to quality; the 20 surveyed companies used different specific practices but all adhered to the principles at the heart of the Baldrige Award and TQM.

But while the GAO report is the best evidence yet of the effectiveness of TQM, it is hardly conclusive. As has been pointed out by critics of the study, the study did not employ statistical methods that would allow firm conclusions to be reached. Furthermore, not all survey questions were answered by all 20 companies. The average number of respondents per question was only 9. Eleven companies did not answer employee satisfaction questions. Supporters of the report's validity attribute the low response rate to company confidentiality requirements, but it is difficult to verify that this is the true reason.

Regardless of the overall question of the effectiveness of TQM and the Baldrige criteria, it is clear that many TQM-compatible quality assurance techniques can produce favorable results. Colson and Prell [Colson 92] describe successful use of TQM on a large software project at AT&T. Murine [Murine 88] reports successful use of software quality assurance methods by Japanese companies like NEC, Mitsubishi, Fujitsu, and Nippon. He claims cost reductions of 30 to 35 percent through the development cycle, and up to 85 percent reduction if resulting improved system maintainability is taken into account.

13.6 Suggestions for Introduction and Use

As was noted earlier, TQM is not "total" unless it is deployed widely, across the entire organization. The literature is quite clear in stating that full benefits will not be realized from a piecemeal approach. Colson and Prell [Colson 92] provide the following advice to those implementing TQM in a software development setting:

- Active support and participation from senior management is essential.
- A system-wide view of process is crucial to establish goals and directions that actually contribute to strategic business planning and satisfied external customers.
- Formal process ownership must reside at the level of management that can implement change.
- It is essential that all levels—from engineers up through senior management—exercise ownership for process improvement.

- Process work is a project—it must be planned, designed, implemented, and managed.
- Process work must have a dedicated staff, both within each process and in a centralized consulting and support organization.
- Communication with the entire development community is crucial to ensure commonality of goals and support of process change.

Briesford [Briesford 88] adds that software quality organizations need:

- visibility,
- supportive management,
- independence from software developers,
- computer-knowledgeable management,
- ability to retain software personnel, and
- a career path for software quality assurance professionals.

Many authors emphasize that TQM is not so much a method or technique as it is a whole new way of doing and thinking about things. As such, it is not to be taken lightly. Only when the management of an organization can wholeheartedly back a dramatic change in their organization should TQM be attempted.

13.7 How TQM Is Related to the Capability Maturity Model

As with QFD, TQM is a method best applied within the level 4 (managed) KPAs, organization process management and software quality management. Some of the concepts could also be applied within the software quality assurance KPA at level 2 of the CMM, and within the organization process focus KPA at level 3 of the CMM. However, extensive application of TQM requires substantial maturity of the software development organization's process. Application of TQM is perhaps the most pervasive strategy of level 3 organizations wishing to achieve levels 4 and 5 of the CMM. The advanced quality control techniques of TQM usually require application beyond the software development organization, i.e., to the business enterprise. The TQM method is best applied within organizations motivated to improve their quality culture.

13.8 Summary Comments on TQM

TQM is widely believed, although not conclusively shown, to yield great benefits to organizations that adopt it. Ironically, its chances for success are perhaps hampered by its rapid rate of acceptance within U.S. businesses. Because so many people are eager to "sell" TQM, to their own organization or to others, it is difficult to sort out pluses and minuses at the level of detail necessary for implementation. We suggest that anyone thinking of implementing TQM seek out not only pro-TQM, but also anti-TQM commentary. Find out, for example, what Deming dislikes about the way TQM is often implemented. As with most attempts to improve the way business is done, the most successful approaches will result from careful adaptation of what already exists to form a coherent set of quality principles suited to the application environment.

13.9 References and Further Readings - TQM

[Anjard 92] Anjard, R. P. Sr., "Software Quality Assurance Considerations," *Electronics Reliability*, Vol. 32, No. 3, 307-312, 1992.

[Basili 87] Basili, V. R., Rombach, H. D., "Implementing Quantitative SQA: A Practical Model," *IEEE Software*, 6-9, September, 1987.

[Breisford 88] Breisford, J. J., "Establishing a Software Quality Program," *Quality Progress*, November, 1988.

[Cavano 87] Cavano, J. P., LaMonica, F. S., "Quality Assurance in Future Development Environments," *IEEE Software*, 26-34, September, 1987.

[Colson 92] Colson, J. S., Prell, E. M., "Total Quality Management for a Large Software Project," *AT&T Technical Journal*, 48-56, May/June, 1992.

[Crosby 92] Crosby, P. B., "Debate," *Harvard Business Review*, 127-128, January-February, 1992.

[Deming 92] Deming, W. E. "Debate," *Harvard Business Review*, 134, January-February, 1992.

[Elmendorf 92] Elmendorf, D. C. "Managing Quality and Quality Improvement," *AT&T Technical Journal*, 57-65, May/June, 1992.

[Garvin 91] Garvin, D. A., "How the Baldrige Award Really Works," *Harvard Business Review*, 80-93, November-December, 1991.

[Ishikawa 85] Ishikawa, K., *What is Total Quality Control? The Japanese Way*, trans. by D. J. Lu, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Kishida 87] Kishida, K., Teramoto, M., Torii, K., Urano, Y., "Quality-Assurance Technology in Japan," *IEEE Software*, 11-17, September, 1987.

[Murine 88] Murine, G. E., "Integrating Software Quality Metrics with Software QA," *Quality Progress*, November, 1988.

[Ross 93] Ross, J. E., *Total Quality Management: Texts, Cases, and Readings*, St. Lucie Press, Delray Beach, FL, 1993.

[USGAO 90] United States General Accounting Office, *Management Practices – U.S. Companies Improve Performance Through Quality Efforts*, May, 1990.

[Zultner 90] Zultner, R. "Software Total Quality Management: What does it take to be World-Class?," *American Programmer*, 1-6, November, 1990.

[Zultner 88] Zultner, R. "The Deming Approach to Software Quality Engineering," *Quality Progress*, 58-64, November, 1988.

14 Defect Prevention Process (DPP)

14.1 Overview

The defect prevention process (DPP) is a means of preventing the insertion of defects into code and other structured software development work products. It is a team technique based on *causal analysis* as introduced by the Japanese quality expert Ishikawa [Ishikawa 85]. Defects are analyzed to uncover root causes and actions are initiated to eliminate causes. The process of defect analysis and prevention is integrated into the overall development process, resulting in lower defect insertion, and consequently higher quality and productivity (due to less rework).

14.2 History/Background

The DPP originated within IBM in the early 1980s, but related causal analysis techniques have been in use, especially in Japan, for much longer [Mays 90]. Wider use of the DPP began in 1985 with the publication of a description of the method by Jones. While there is evidence that the DPP has been used successfully within IBM ([Kolkhorst 88], [Spector 84], [Gale 90]), the base of documented experience with the technique remains limited. The successful reports thus far, and the obvious success of related techniques in nonsoftware design and production, provide reason for cautious optimism concerning the potential widespread benefits of defect prevention.

14.3 Description of the Defect Prevention Process

The DPP calls for integrating into each stage of software development two additional activities, the *kickoff meeting* and the *causal analysis*, and also the addition of one organizational entity, the *action team*. The process is integrated into an entry-task-validation-exit (ETVX) framework.

The kickoff meeting is added to the entry substage. At this meeting, the development team reviews the inputs from the previous stage, for completeness and also to promote an understanding of the task at hand. Process and methodology guidelines are reviewed, and developers are exposed to a list of common errors for that stage of development (this list is derived from the DPP in a way that will become apparent). Also at the kickoff meeting, the team sets goals for error prevention and detection. Team goals are for the team's own internal use and are never published.

Causal analysis is added to both the validation and exit substages. During validation, formal inspections (see Chapter 7) are conducted and defects are detected. The DPP enhancement is to have the author of discovered defects add them to a database, along with the defect resolution information and a preliminary analysis of what caused the defect. In the exit substage, a causal analysis meeting is conducted. Defects are analyzed to determine their causes and the stage of development in which they originated. Causes are classified as communications (i.e., failure of), education (i.e., a team member fails to understand something), oversight (i.e., some possible conditions are overlooked), and transcription (e.g., a typo). Team members suggest ways of avoiding each defect and recommend corrective actions to prevent similar such errors in the future.

The action team is a group within the organization with responsibility for acting on the recommendations that result from causal analysis. The action team prioritizes, then implements and tracks action items. They are also responsible for reporting back actions taken to the development community, developing common error lists for use in kickoff meetings, administering the defect prevention database, and suggesting action items derived from analysis of problems with broad trends across projects.

14.4 Experience with Defect Prevention

Documented experience with the DPP comes exclusively from within IBM, in limited quantities. However, Mays et al. [Mays 90] report that the process has been used "in more than 25 organizations at 7 IBM development laboratories, involving systems programming, application programming, and microcode development."

Accounts of the results of the use of defect prevention suggest that the process results in considerably lower defect insertion rates. One product studied in detail by Mays et al. [Mays 90] showed a reduction in defects averaging 54 percent when the DPP was introduced for later releases. The costs of the DPP introduction were largely due to the additional time required for DPP meetings and for following up on action items; total costs, according to Mays et al. [Mays 90], are between 0.4 and 0.5 percent of total project resources. Benefits are harder to quantify but arise due to less time spent detecting, investigating, and fixing defects, and would seem quite substantial.

The potential long-term benefits are great in that the causal analysis, with its emphasis on attacking root causes, can potentially lead to the "extinction" [Mays 90] of whole classes of errors. Causal trends can be identified and warnings to developers placed in strategic locations (e.g., checklists, process documentation, tools that make automatic checks, templates for work products). Also, substantial process improvements to the overall development may result as the real root causes of problems become apparent. The DPP is a technique for substantially reducing the defect insertion rate for a software development process. Since every defect has an associated cost for detection and correction, the cost reduction benefits of DPP are apparent when fewer defects are introduced into the software development process.

14.5 Suggestions for Introduction and Use

Mays et al. [Mays 90] provide the following suggestions for introduction and use of defect prevention.

- Identify a sponsor in a high-level position willing to be an advocate for the process; ideally two advocates would be identified, one a technical person and the other a manager.
- Advocates should start by educating managers and developers about the DPP, on cost and benefits, etc.
- Advocates should actively supervise initial DPP efforts, to assure that start-up efforts are not undermined by problems of logistics, lack of focus, etc.
- Start small, with a pilot project, and then gradually expand use of the process.
- Action team members should be carefully chosen; they should be highly motivated and dedicated to improving the area's processes.
- Action team members should strive for quick implementation of some early action items, to establish the credibility of the process.
- Early kickoff and causal analysis meetings should include a significant amount of instruction on how to carry out the DPP.
- Be alert in early meetings for an initial sensitivity and defensiveness because developers are being asked to analyze their own mistakes; defensiveness can be diffused by a skillful moderator who should work to keep the atmosphere of the meeting nonthreatening.

14.6 How Defect Prevention Is Related to the Capability Maturity Model

Defect prevention is an elaboration on the inspection process and requires that an inspection-like means of error detection be in place. The DPP process is, essentially, a means of tracking down sources of variation in the software development process and taking action to prevent future injection of defects. As such, it is located at level 5 of the Capability Maturity Model (the optimizing level), as the defect prevention KPA. Less formal versions of defect prevention may, however, reasonably arise as early as level 2 (the repeatable level), since ensuring repeatability of the development process may require at least rudimentary causal analysis of problems.

It is also interesting to speculate when an organization could consider DPP as related to their process maturity. In general, it is easier to *detect* defects effectively before one has the knowledge to *prevent* defects. Thus, peer reviews as implemented by formal inspections (level 3) are a necessary prerequisite for DPP application. There is also a dependency on the KPA process measurement and analysis (level 4) since many defects are caused by deficiencies of the software development process, and quantitative evidence and analysis may be needed to identify and confirm these deficiencies. The action team will need to rely on process measurements (see Chapter 8) to identify the appropriate process improvement actions to implement. Thus, a level 3 organization with process metrics can reasonably begin the planning necessary to introduce DPP.

14.7 References and Further Readings - DPP

[Gale 90]	Gale, J. L, J. R. Tirso, C. A. Burchfield, "Implementing the Defect Prevention Process in the MVS Interactive Programming Organization," *IBM Systems Journal,* Vol. 29, No. 1, 1990, pp. 33-43.
[Ishikawa 85]	Ishikawa, K., *What is Total Quality Control? The Japanese Way,* translated by D. J. Lu, Englewood Cliffs, NJ: Prentice Hall, 1985.
[Jones 85]	Jones, C. L. "A Process-Integrated Approach to Defect Prevention," *IBM Systems Journal,* Vol. 24, No. 2, 1985, pp. 150-167.
[Kolkhorst 88]	Kolkhorst, B. G., A. J. Macina, "Developing Error-Free Software," *IEEE Aerospace Electronic Systems Magazine,* Vol 3. No. 11, Nov. 1988, pp. 25-31.
[Mays 90]	Mays, R. L, C. L. Jones, G. J. Holloway, D. P. Studinski, "Experiences with Defect Prevention," *IBM Systems Journal,* Vol. 29, No. 1, 1990, pp. 4-32.
[Spector 84]	Spector, A., Gifford, D. "The Space Shuttle Primary Computer System," *Communications of the ACM,* Vol. 27, No. 9, 1984, pp. 874-900.

15 Cleanroom Software Development

15.1 Overview

Cleanroom software development is a software production method that emphasizes preventing the introduction of errors rather than testing to remove errors after they have been introduced. Cleanroom combines formal specification, nonexecution-based program development, incremental development, and independent statistical testing. Of these features, nonexecution-based program development is the point of greatest departure from more traditional methods. In cleanroom, development teams literally do not have access to compilers to produce executable code, so they cannot rely on automated checking to identify errors. Realizing this, the team develops a mind set that facilitates prevention of defects. Error rates in cleanroom software are usually lower than error rates in software produced by more traditional methods. The goals of cleanroom are to reduce life-cycle costs by reducing rework and to produce certifiably reliable software.

15.2 History/Background

Cleanroom originated in the Federal Systems Division of IBM in the late 1970s and early 1980s [Dyer 92]. It has not gained wide acceptance, perhaps because it represents a dramatic change from the traditional way of developing software. Nonexecution-based program development probably sounds odd to organizations investing in ever more advanced personal computer and workstation based program editors and compilers. But where cleanroom has been used, results have been impressive. In contrast with many of the methods described in this report, there have been controlled studies of cleanroom that show it is superior to more traditional methods on a variety of dimensions, at least in experimental settings. As with any relatively new method, problems with cleanroom may become apparent as it becomes more widely used. But even at this early stage of use, claims of the advantages of cleanroom seem better supported than claims about some other methods that are more widely deployed.

15.3 Description of Cleanroom Software Development

Cleanroom has four essential features that set it apart from more conventional methods.

1. *Incremental development:* Cleanroom calls for incremental development rather than the sequential process of analysis, design, implementation and testing used in traditional development processes. A system with bare bones functionality is constructed and then the functionality

is extended incrementally. Each successive release adds functionality. Functionality accumulates as development progresses and finally results in a fully functional product.

2. *Formal methods:* Cleanroom uses formal methods for specification and design. Unlike other development methodologies, which may also use formal methods, Cleanroom depends on the formal methods to divide system functionality into deeply nested subsets that can be developed incrementally [Selby 87]. The formal methods make use of black boxes, state boxes, and clear boxes, each of which specify the system functionality at a lower level of detail. The level of detail called for by each is rigorously defined. For example, the black box "is a precise specification of external, user-visible behavior in all possible circumstances of use" [Linger 92a]. The precise specifications allow independent testing of the product at each incremental level of detail.

3. *Development without program execution:* Developers are literally restricted from using any tool that would allow testing of the executability of the code. Developers do not participate in testing and debugging at all. Instead, they focus on code inspections and other "off-line software review techniques" [Selby 87] to assure the correctness of their implementation. Without the crutch of execution-based testing, developers are forced to produce coherent, easy-to-inspect designs and code, and to assure their correctness by close attention to the task at hand. The mind set of developers becomes changed when they are freed from the assumptions that "we can always catch errors in testing."

4. *Independent statistically-based testing:* The independent group that performs testing adopts a perspective of reliability assessment (see Chapter 11, which covers SRE) rather than error detection and elimination. The reliability group runs the software with test cases from probability distributions across all possible user inputs and system states, at each incremental stage of development. Special test cases are designed to head off catastrophic failures. The end result of the testing is a certified level of reliability in terms of "mean time to failure" or some similar reliability measure.

Thus, cleanroom incorporates many of the other software process improvement methods described in this report such as formal inspection, CASE, measurement, SRE, DPP, etc.

15.4 Experience with Cleanroom Software Development

Most of the documented experience with cleanroom has occurred in or in conjunction with IBM's Federal Systems Division. Reports of remarkably low error rates are not uncommon [Mills 91].

Hevner et al. [Hevner 92] report results from seven full and partial cleanroom projects between 1980 and 1990. In two products produced by these projects, no defect had ever been found during customer use. One of these two products consists of 25 KLOC and the other is 63 KLOC, and both have been in operation since the early 1980s.

Linger and Hausler [Linger 92a] report results from 11 full and partial cleanroom projects between 1987 and 1992. Two of these were among those reported by Hevner et al. Many of these projects cited productivity gains. That there were savings from foregone maintenance is obvious.

Selby et al. [Selby 87] conducted a controlled experimental study of the effectiveness of the method. They compared 10 cleanroom teams with 5 non-cleanroom teams working for six weeks on exactly the same roughly 1500 line application. Subjects were computer science students at the University of Maryland who averaged 1.6 years of professional experience and had graduate, senior, or junior standing. There were three students on each team. Researchers used "Simpl-T," a language unknown to all subjects at the start of the experiment. Use of an obscure language eliminated experience with language confounds and enforced the prohibition against use of compilers by cleanroom teams, because no compiler for Simpl-T was available. Other factors that might have produced confounds (such as academic performance) were divided across teams as evenly as possible. This research produced seven major results:

1. Six of 10 cleanroom teams delivered at least 91 percent of the required system functions, while only one of the five non-cleanroom teams did.
2. Cleanroom team products met system requirements more completely and had a higher percentage of successful, operationally-generated test cases.
3. Source code developed using cleanroom had more comments and less dense control flow complexity.
4. The more successful cleanroom teams modified their use of the implementation language; they used more procedure calls and IF statements, fewer CASE and WHILE statements, and had a lower frequency of variable reuse.

5. All 10 cleanroom teams made all of their scheduled intermediate product deliveries, while only two of the five non-cleanroom teams did.

6. Eighty-six percent of the cleanroom developers indicated that they missed the satisfaction of program execution to some extent, but this had no relationship with product quality.

7. Eighty-one percent of the cleanroom developers said they would use the method again.

Some of the qualitative results of the Selby et al. study do suggest areas of concern, however. For example, several people from cleanroom teams said they had difficulty "visualizing the user interface," and therefore felt that the system might not be user-friendly enough. Nonetheless, the incremental design method should facilitate user interaction. It is possible that with slight modification, the cleanroom method might solve this problem. Also, as Selby et al. note, the method could be easily combined with a prototyping approach, which would provide a user interface early in a project.

Of other possible issues, the most important is surely one having to do with culture change. Developers are not likely to give up their favorite editor/compiler/debugger easily. Many developers got started in the profession because of a love of interacting with the computer, making changes, and watching their effects. Cleanroom may be seen by these developers as an unattractive way to develop software. Those who resist development without program execution may find ways to gain access to compilers despite whatever precautions are taken against it. Real applications cannot be developed in Simpl-T or some similarly obscure language. Selby, et al. propose that other methods of satisfying the desire to see program execution might be invented. They suggest having the development team watch system testing. This idea seems promising when one envisions anxious developers gathered around the testing team to cheer for their software.

We know of no exhaustive study that compares costs of traditional development with cleanroom. Such a test would be difficult because cleanroom would gain much of its savings from foregone maintenance due to much lower error rates; such savings necessarily are realized over time. But the low error rates associated with cleanroom suggest that savings of this kind could be considerable. Also, from the Selby et al. study it is apparent that cleanroom need not cost more in the early stages of development. In the experiment, cleanroom teams consistently met project schedules and delivered functionality where teams using more traditional methods did not. Linger and Spangler [Linger 92b] claim that quality improvements offset all introduction costs, and this does not seem unreasonable.

15.5 Suggestions for Introduction and Use

Cleanroom is an advanced method of developing computer software. The method itself requires training, and the testing capabilities of the organization seeking to implement cleanroom must be well established and sophisticated. Testing from a reliability perspective requires that the organization have reliability testing up and running. Furthermore, rigorous formal specification requires training and considerable process sophistication. Inspections and other "off-line software review techniques" should already be in place. Because of all of these factors, cleanroom is among the most difficult of the methods in this report to implement. Attempts to implement cleanroom should perhaps be confined to development teams that are particularly adept, or deployed widely in very mature software organizations.

For those that do attempt it, however, there is advice available. IBM's Cleanroom Software Technology Center (CSTC) provides training and other cleanroom-related services. Their program for getting a cleanroom project started is evidence of the complexity of the method [Linger 92b].

- Initial planning must include selection of an appropriate project, reviews of staffing and resource plans, and management briefings; setting expectations and defining success factors is the critical function of this stage.
- Education of the cleanroom team involves a one day introductory class and at least three longer and more advanced classes.
- First attempts at cleanroom benefit from close consultation with someone experienced in the method.
- Propagating cleanroom throughout an organization is best accomplished by assigning original cleanroom team members to work with or head other teams that want to attempt cleanroom development.

15.6 How Cleanroom Software Development Is Related to the Capability Maturity Model

Cleanroom is perhaps the most sophisticated and complex software process improvement method described in this report. Organizations that implement cleanroom exhibit many of the characteristics of levels 4 and 5 of the CMM. For these reasons, it is recommended that only organizations currently performing at levels 3 or 4 of the CMM consider implementing cleanroom. Cleanroom most closely correlates to the level 4 key process areas, quantitative process management and software quality management, and to the level 5 key process area, defect prevention.

15.7 Summary Comments on Cleanroom Software Development

Cleanroom seems to hold great promise. It is not technology intensive and therefore is not costly in that sense. It seems likely to reduce overall system life-cycle costs. However, it requires considerable organizational maturity of those who seek to implement it. It is also quite new. As with all new methods, problems may become apparent as more experience with the method is accumulated. For the mature software organization willing to pioneer a new development process, though, cleanroom seems worth a try.

15.8 References and Further Readings - Cleanroom

[Currit 86] Currit, P. A., Dyer, M., Mills, H.D., "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, January, 1986.

[Dyer 87] Dyer, M., "A Formal Approach to Software Error Removal," *The Journal of Systems and Software*, Vol. 7, 109-114, 1987.

[Dyer 92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, J. Wiley & Sons, 1992.

[Hevner 92] Hevner, A. R., Vagoun, T., Lemmon, D., "Quality Measurements in the Cleanroom Development Process," *Proceedings of the Second International Conference on Software Quality*, Research Triangle Park, NC, October 5-7, 1992.

[Linger 92a] Linger, R. C., Hausler, P. A., "On the Road to Zero Defects with Cleanroom Software Engineering," IBM Technical Report, 1992.

[Linger 92b] Linger, R. C., Spangler, R. A., "The IBM Cleanroom Software Engineering Technology Transfer Program," *Proceedings of the Sixth SEI Conference on Software Engineering Education*, San Diego, CA, October 5-7, 1992.

[Mills 91] Mills, H.D., "Cleanroom Engineering: Engineering Software Under Statistical Quality Control," *American Programmer*, May 1991.

[Selby 87] Selby, R. W., Basili, V. R., Baker, F. T., "Cleanroom Software Development: An Empirical Evaluation," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 19, September, 1987.

16 Conclusions

This literature survey indicates that there is substantial evidence that industrial software development organizations have been successful in applying the described process improvement methods. The methods have had a positive result on the organizations' ability to produce higher quality software products, improve productivity, or reduce development cycle times.

The major variable of application success appears to be the ability of the organization to adapt to the change that the method introduces to the current work habits. The variance in the implementation difficulty of the methods appears to be quite large. One may categorize two classes of methods:

- Those that are procedurally well defined and map to usually one KPA (e.g., formal inspection, software process assessment).
- Those that have multiple approaches dependent on organization maturity and other environmental factors, and that often map to multiple KPAs at various levels of the CMM (e.g., measurement, CASE).

The simpler methods tend to supply more of a "recipe" for application. They are thus easier to implement, and the literature contains generally more experience reports claiming successful implementation and benefits. The more complex methods are often collections of methods that can be applied to many organizational situations, thus increasing their implementation difficulty. In almost all cases, the methods require some degree of "tailoring" to be successfully applied due to the great variance found in industrial software development processes, tools, and environments.

The scope of the methods also varies greatly with respect to the impact on the organization implementing a specific method or collection of methods. Some of the methods described can be applied to specific phases of the development process by isolated functions within the organization, while some methods involve greater numbers of staff and multiple or all phases of the development process. If we consider the relationships of characteristics of a software development organization's product, process, and environment, then we can illustrate the scope of the described methods as shown in Figure 16-1. In general, the methods with smaller scope are easier to implement, but perhaps do not result in benefits as great as the broader scope methods.

The selection of the "best" methods for an organization to apply is largely dependent on the organization's process maturity level, and the implementation difficulty of the method. It is apparent that level 1 organizations should be primarily focussed on documenting their process, and that they should resist attempting the methods that would be more appropriate for level 3-5 organizations. Similarly, one may define a "core" set of methods that does not require a high degree of process maturity and is relatively simple to implement. It is apparent that formal inspection would fall within such a core set of methods since it is well defined, has a large number of practitioners, and can be introduced at any stage of the development process. Similarly, one could apply software process assessment as a diagnostic tool to any software organization as a recommended step towards initiating and sustaining software process improvement activities.

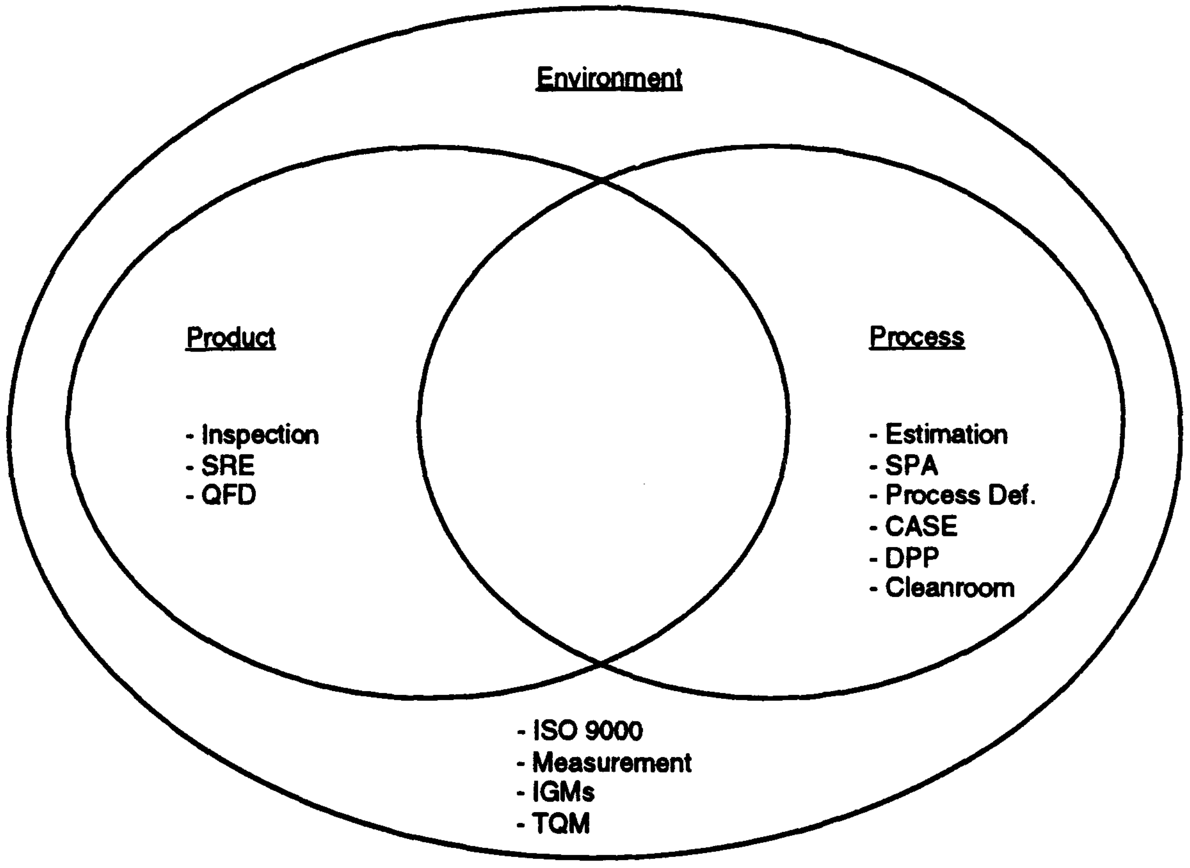

Figure 16-1. Characteristics of a Software Development Organization

Table 4 summarizes some of the implementation issues of the methods described in this report. For each method, we indicate a maturity level range at which an organization should be in order to consider implementing the method for the first time. In addition, a small number of summary pros and cons are indicated concerning implementation of the method.

Table 4: Implementation Issues Sun.₁ıary

Method	Recommended Maturity Level	Pros	Cons
Estimation	1	Fundamental to project planning	Works best when historical data are available
ISO 9000	1	Required for many markets	Emphasis is on evidence rather than improvement
SPA	1-2	Good first step towards process improvement	Investment provides primarily findings
Process definition	1-2	Provides baseline for improvement	Representation tools skills often missing
Formal inspection	1-2	Easy to begin	More commonly used for code than documents
Measurement	1-2	Used with other methods	Must be tailored to goals
CASE	1-2	Automates process	High investment (e.g., license fees, training)
IGMs	1-2	Promote better teamwork	Possible communication & meeting overhead
SRE	2-3	Provides field defect rate predictions	Training and skills often missing
QFD	2-3	Helps build the "right" products	Difficult to manage product complexity and communications
TQM	2-3	Builds a "quality culture"	Requires commitment & implementation throughout organization
DPP	3-4	Makes classes of errors extinct	Can only be considered by mature organizations
Cleanroom	3-4	Can result in high product quality	Different than current development practices

Although measurement is identified as a specific method, its application varies considerably depending on the maturity level of the organization. It may also be viewed as an "accompanying technology" since the measurement data analysis must be used to stimulate other actions for process improvement. Measurement by itself has no significant impact on organization performance. Furthermore, it is strongly suggested that measurement be used in conjunction with all of the described methods. Measurement provides a feedback mechanism to the organization introducing process improvement methods such that it can determine how effective the method is being applied. This feedback will help an organization determine how well the process improvement method is being introduced and accepted, and it will help identify when the organization is ready to introduce additional methods.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-93-TR-27	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESC-TR-93-201
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (city, state, and zip code) HQ ESC/ENS 5 Eglin Street Hanscom AFB, MA 01731-2116
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESC/ENS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 63756E PROJECT NO.: N/A TASK NO: N/A WORK UNIT NO.: N/A

11. TITLE (Include Security Classification)
A Survey of Commonly Applied Methods for Software Process Improvement

12. PERSONAL AUTHOR(S)
Robert D. Austin and Daniel J. Paulish

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	February 1994	126 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
capability maturity model
literature survey
process improvement methods
software process

19. ABSTRACT (continue on reverse if necessary and identify by block number)

This report describes a number of commonly applied methods for improving the software development process. Each software process improvement method is described by surveying existing technical literature citations. Each method description contains background information concerning how the method works. Documented experience with the method is described. Suggestions are given for implementing the method, and a list of key references is given for further information. The methods are described in the context of the SEI Capability Maturity Model, and suggestions are given to assist organizations in selecting potential improvement methods based upon their current process maturity.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF	22b. TELEPHONE NUMBER (include area code) (412) 268-7631 22c. OFFICE SYMBOL ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19